LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/RSS 5
Research Seminar Series

DISTRIBUTED ALGORITHMS

Lecture Notes for 6.862
Fall 1988

Nancy A. Lynch
Kenneth J. Goldman

May 1989

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Distributed Algorithms

Lecture Notes for 6.852

Fall Semester, 1988

Nancy A. Lynch
Kenneth J. Goldman

Contents

Preface

The MIT subject *6.852 Distributed Algorithms* is a graduate level introduction to the theory of distributed computing. Students taking the course are assumed to have a substantial background in both mathematics and computer science. Course material is drawn from many of the important research papers. An emphasis is placed on formal techniques for: defining problems (correctness conditions), describing algorithms, and constructing correctness proofs. Also, several important impossibility results are presented. As there is not time in one semester to cover all the important papers in the area, the material varies from year to year.

Two sets of references are included: The course syllabus, which appears before Lecture 1, is a list of papers organized by topic. An alphabetized bibliography follows Lecture 26. To assist readers in finding a particular topic, we have provided an alphabetic index in addition to the table of contents.

This set of notes was written by students attending the course in the Fall Semester of 1988. Although we have tried to check the notes carefully, there are likely to be errors and omissions, particularly with regard to the references. Readers finding errors in the notes are encouraged to notify us by electronic mail (`bug-6852@tds.lcs.mit.edu`) so that later versions may be corrected.

Acknowledgements

This compilation of notes would not have been possible without the hard work of the following students: Boaz Ben-Zvi, Azer Bestavros, Chris Colby, Mike Eisenberg, Jeff Fried, Sanjay Ghemawat, John Keen, Magda Nour, Jeff Palmucci, Mike Parker, Sharon Perl, Steve Ponzio, Jon Riecke, Atul Shrivastava, Rick Stille, Andrew Sutherland, Mark Tuttle, and George Varghese.

We thank Hagit Attiya and Mark Tuttle for their "guest" lectures, and Alan Fekete for his help in selecting papers for the course.

Finally, we thank Laura Sprung and Anna Wiseman for their help in preparing the course syllabus and bibliography.

Bibliography

[1] L. Lamport and N. Lynch *Chapter on Distributed Computing.* To appear in *Handbook of Theoretical Computer Science.*

I. Shared Memory

A. Mutual Exclusion

[2] M. Raynal. *Algorithms for Mutual Exclusion.* M.I.T. Press, 1986.

[3] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8:569, 1965.

[4] D. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[5] J. DeBruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, 1967.

[6] M. Eisenberg and M. McGuire. Further comments on Dijkstra's concurrent programming control. *Communications of the ACM*, 15(11):999, 1972.

[7] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of 18th Annual Allerton Conference on Communications, Control, and Computing*, pages 833–842, 1980.

[8] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[9] L. Lamport. The mutual exclusion problem. *Journal of the ACM*, 33(2):313–326, 327–348, 1986.

[10] G. Peterson and M. Fischer. Economical solutions for the critical section problem in a distributed syste. In *Proceedings of 9th ACM Symposium on Theory of Computing*, pages 91–97, May 1977.

[11] J. Burns, M. Fischer, P. Jackson, N. Lynch, and G. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, 1982.

[12] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to limited process failure. In *Proceedings of 20th IEEE Symposium on Foundations of Computer Science*, pages 234–254, October 1985.

[13] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Distributed FIFO allocation of identical resources using small shared space. MIT/LCS/TM-290, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA 02139, October 1985.

[14] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: l-exclusion as a test case. In *Proceedings of 20th ACM Symposium on Theory of Computing*, pages 78–92, May 1988.

[15] M.O. Rabin. N-process mutual exclusion with bounded waiting by $4 \log N$ shared variable. *Journal of Computer and Systems Sciences*, 25:66–75, 1982.

[16] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, 1978.

[18] G. Ricart and A.K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981. Corrigendum in *Communications of the ACM*, 24(9).

[19] O. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–148, 1983.

B. Dining Philosophers

[20] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 115–138, 1971.

[21] N. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal Of Computer And Systems Sciences*, 23(2):254–278, October 1981.

[22] K. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[23] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, pages 133–138, 1981.

C. Modelling

[24] N. Lynch. I/O automata: a model for discrete event systems. Technical Memo MIT/LCS/TM-351, Massachusetts Institute Technology, Laboratory for Computer Science, March 1988. Also, in *Proceedings of 22nd Annual Conference on Information Science and Systems*, March 1988.

[25] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

D. Shared Registers

[26] G. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[27] L. Lamport. On interprocess communication. *Distributed Computing*, 1(1):77–85, 86-101, 1986.

[28] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 249–259, August 1987. Also, to appear in special issue *IEEE Transactions On Computers.*

[29] P.M.B. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings of 27th Annual IEEE Symposium on Theory of Computing*, pages 233–243, May 1986. Also, MIT/LCS/TM-314, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., 1986. Corrigenda in *Proceedings of 28th Annual IEEE Symposium on Theory of Computing*, page 487, 1987.

[30] R. Schaffer. On the correctness of atomic multi-writer registers. Technical Memo MIT/LCS/TM-364, Massachusetts Institute Technology, Laboratory for Computer Science, 1988.

[31] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[32] M. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

II. Consensus

A. Basic Results

[33] J. Gray. *Notes on data base operating systems.* Technical Report IBM Report RJ2183(30001), IBM, February 1978. (Also in Operating Systems: An Advanced Course, Springer-Verlag Lecture Notes in Computer Science #60.).

[34] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[35] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[36] D. Dolev and H.R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM J. Computing*, 12(4):656–666, November 1983.

[37] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[38] R. Turpin and B. Coan. Extending binary Byzantine agreement to multivalued byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984. Also, revised in B. Coan, *Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

B. Number of Processes

[39] D. Dolev. The Byzantine generals strike again. *J. Algorithms*, 3:14–30, 1982.

[40] M. Fischer, N. Lynch, and M. Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1:26–39, 1986.

[41] L. Lamport. The weak Byzantine generals problem. *Journal of the ACM*, 30(3):669–676, 1983.

[42] D. Dolev, J. Halpern, and R. Strong. On the possiblity and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32:230–250, 1986.

C. Time

[43] M. Fischer and N. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, 1982.

[44] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment I: crash failures. In *Proceedings of Conference on Theoretical Aspects of Reasoning about Knowledge*, 1986. Also, to appear in *Information and Computation.*

[45] Y. Moses and M. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:249–259, 1988.

D. Polynomical Communication

[46] B.A. Coan. A communication-efficient canonical from for fault-tolerant distributed protocols. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 63–72, August 1986. Also, revised in B. Coan, *Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[47] A. Bar-Noy, D. Dolev, C. Dwork, and H. R. Strong. Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 42–51, August 1987.

[48] Y. Moses and O. Waarts. Coordinated Traversal: $(t+1)$-round byzantine agreement in polynomial time. In *Proceedings of 29th Symposium on Foundations of Computer Science*, pages 246–255, October 1988.

E. Asynchronous Results

[49] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one family faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[50] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[51] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[52] J.L. Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, 1987.

[53] S. Moran and Y. Wohlfstahl. Extended impossibility results for asynchronous complete networks, *Information Processing Letters*, 26:145–151, 1987.

[54] M.F. Bridgeland and R.J. Watro. Fault tolerant decision making in totally asynchronous distributed systems. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987.

[55] O. Biran, S. Moran, and S. Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *proceedings of 7th ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[56] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. Manuscript

F. Randomized Algorithms

[57] B. Chor and B.A. Coan. A simple and efficient randomized byzantine agreement algorithm. *IEEE Transactions on Software Engineering*, SE-11:531–539, 1985. Also, revised in B. Coan, *Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[58] M. Rabin. Randomized *Byzantine* generals. In *Proceedings of 24th IEEE Symposium on Foundations of Computer Science*, pages 403–409, November 1983.

[59] G. Bracha. An $O(\log n)$ expected rounds randomized *Byzantine* generals algorithm. *Journal of the ACM*, 34(4):910–920, 1987.

[60] P. Feldman. Optimal Byzantine agreement. Ph.D. thesis, Department of Mathematics, Massachusetts Institute of Technology, 1988.

G. Approximate Agreement

[61] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):449–516, 1986.

H. Partial Synchrony

[62] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

I. Clock Synchronization

[63] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, 1978.

[64] L. Lamport and P. Melliar-Smith. Byzantine clock synchronization. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 68–74, August 1984.

[65] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[66] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2-3):190–204, 1984.

[67] J. Halpern, B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 89–102, August 1984.

[68] S. Mahaney and F. Schneider. Inexact agreement: accuracy, precision, and graceful degradation. In *Proceedings of 4th ACM Symposium on Principles of Distributed Computing*, pages 237–249, August 1985.

J. Firing Squad

[69] J. Burns and N. Lynch. The Byzantine firing squad problem. Technical Memo MIT/LCS/TM-275, Laboratory for Computer Science,Massachusetts Institute Technology, April 1985.

[70] B. Coan, D. Dolev, C. Dwork, and L. Stockmeyer. The distributed firing squad problem. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 335–345, May 1985.

K. Commit

[71] C. Dwork and D. Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, August 1983.

[72] B. Coan and J. Lundelius. Transaction commit in a realistic fault model. In *Proceedings of 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 40–51, August 1986.

III. Knowledge

[73] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 50–61, 1984. Revised as IBM Research Report, IBM-RJ-4421.

[74] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment I: crash failures. In *Proceedings of Conference on Theoretical Aspects of Reasoning about Knowledge*, 1986. Also, to appear in *Information and Computation.*

[75] Y. Moses and M. R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:249–259, 1988.

[76] K.M. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1:40–52, 1986.

IV. Static Network Algorithms

A. Symmetry

[77] D. Angluin. Local and global properties in networks of processors. In *Proceedings of 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.

B. Computing in a Ring

[78] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, 1977.

[79] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, 1979.

[80] D.S. Hirschberg and J.B. Sinclair. Decentralized extrema-finding in circular configuarations of processes. *Communications of the ACM*, 23:627–628, 1980.

[81] G.L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4:758–762, 1982.

[82] D. Dolev, M. Klawe, and M. Rodeh. An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *J. Algorithms*, 3:245–260, 1982.

[83] J. Burns. *A formal model for message passing systems.* Technical Report TR-91, Computer Science Dept., Indiana University, May 1980.

[84] G.N. Frederickson and N. Lynch. Electing a leader in a synchronous ring. *Journal Of The ACM*, 34(1):98–115, January 1987. Also, MIT/LCS/TM-277, July 1985.

[85] H. Attiya, M. Snir, and M. Warmuth. Computing in an anonymous ring. To appear in *Journal of the ACM.*

[86] K. Abrahamson, A. Adler, L. Higham, and D. Kirkpatrick. *Probabilistic Solitude Detection I: Ring Size Known Approximately.* Technical Report 87-8, University of British Columbia, Vancouver, B.C., Canada, March 1987.

[87] K. Abrahamson, A. Adler, L. Higham, and D. Kirkpatrick. *Probabilistic Solitude Detection II: Ring Size Known Exactly.* Technical Report 87-11, University of British Columbia, Vancouver, B.C., Canada, April 1987.

C. Complete Graphs

[88] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 199–207, 1984.

[89] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of 4th ACM Symposium on Distributed Computing*, pages 186–195, August 1985.

D. Arbitrary Graphs

[90] R. Gallager, P. Humblet, and P.A. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, 1983.

[91] P. Humblet. A distributed algorithm for minimum weight directed spanning trees. *IEEE Transactions on Communication*, COM-31(6):756–762, 1983.

[92] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. A tradeoff between information and communication in broadcast protocols. In *Proceedings of the Aegean Workshop on Computing*, 1988.

V. Dynamic Network Algorithms

A. Synchronization

[93] E. Arjomandi, M. Fischer, and N. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the Association for Computing Machinery*, 30(3):449–456, 1983.

[94] B. Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, 1985.

[95] B. Awerbuch. Reducing complexities of distributed maximum flow and breadth-first search algorithms by means of network synchronization. *Networks*, 15:425–437, 1985.

B. Termination Detection

[96] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), 1980.

[97] K. M. Chandy and J. Misra. On proofs of distributed algorithms, with application to the problem of termination detection. Manuscript.

C. Global Snapshots

[98] K.M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[99] M. Fischer, N. Griffeth, and N. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, 1982.

D. Deadlock Detection

[100] D.A. Menasce and R.R. Muntz. Locking and deadlock detection in distributed databases. *IEEE Transactions on Software Engineering*, SE-5(3):195–202, 1979.

[101] V.D. Gligor and S.H. Shattuck. On deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435–439, 1980.

[102] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, 1982.

[103] K.M. Chandy, J. Misra, and L. Haas. Distributed deadlock detection. *ACM Transactions on Programming Languages and Systems*, 1(2):144–156, 1983.

[104] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[105] D. Mitchell and M. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of 3rd ACM Symposium on Distributed Computing*, pages 282–284, August 1984.

E. Fault-Tolerant Computation

[106] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1), pages 23–35, 1983.

[107] A. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proceedings of 28th IEEE Symposium on Foundations of Computer Science*, pages 358–370, October 1987.

[108] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Proceedings of 29th IEEE Symposium on Foundations of Computer Science*, October 1988.

F. Communication

[109] A.V. Aho, J.D. Ullman, A.D. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982.

[110] N. Lynch, Y. Mansour, and A. Fekete. The data link layer: two impossibility results. In *Proceedings of 7th ACM Symposium on Principles of Distributed Computing*, pages 149–170, August 1988.

VI. Concurrency Control

A. Classical Theory

[111] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1986.

[112] A. ElAbbadi and S. Toueg. Maintaining availability in partitioned replicated databases. In *Proceedings of 5th ACM Symposium on Principles of Database Systems*, pages 240–251, 1986.

B. Our Theory

[113] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. Technical Memo MIT/LCS/TM-370, Massachusetts Institute Technology, Laboratory for Computer Science, 1988.

[114] J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of 14th International Conference on Very Large Data Bases*, page 431–444, August 1988.

[115] K. Goldman and N. Lynch. Quorum consensus in nested transaction systems. In *proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 27–41, August 1987. Also, extended version in MIT/LCS/TR-390, MIT Laboratory for Computer Science, Cambridge, MA, May 1987.

[116] M. Herlihy, N. Lynch, M. Merritt, and W. Weihl. On the correctness of orphan elimination algorithms. In *Proceedings of 17th IEEE Symposium on Fault-Tolerant Computing*, pages 8–13, 1987. Also, revised version to appear in *Journal of the ACM.*

C. Nonserializable Databases

[117] M. Fischer and A. Michael. *Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network.* Research Report 221, Yale University, February 1982.

[118] S.K. Sarin and N. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering*, SE-13(1):39–47, 1987.

[119] N. Lynch, B. Blaustein, and M. Siegel. Correctness conditions for highly available replicated databases. In *Proceedings of 5th ACM Symposium on Principles of Distributed Computing*, pages 11–28, August 1986.

VII. More on Modelling and Proofs

[120] N. Lynch and M. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[121] C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1984.

[122] S. Owicki and D. Gries. An axiomatic proof technique for parrallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[123] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, 1983.

[124] K. M. Chandy, and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, 1988.

[125] M. Staskauskas. The formal specification and design of a distributed electronic funds-transfer system. *IEEE Transactions on Computers*, 37(12):515–528, December 1988.

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 1: September 13

Lecturer: Nancy Lynch *Scribe: Christopher Colby*

1.1 Introduction

1.1.1 Characteristics of distributed algorithms

Distributed algorithms include a wide range of parallel algorithms; for example, shared-memory algorithms, message-passing algorithms, synchronous algorithms, asynchronous algorithms, dataflow algorithms, and database algotithms.

Not all parallel algorithms, however, are distributed algorithms. Some algorithms involve *tightly-coupled* parallelism, where all of the components are cooperating to solve one problem. In contrast, distributed algorithms are characterized by the *independence* of activities and *loosly-coupled* parallelism:

- independent inputs with outputs at multiple locations
- several programs executing simultaneously under separate control
- processes starting at different times
- failures of processes
- processes running at different speeds
- uncertainty—each process is aware of only a part of the state of the system

1.1.2 The study of distributed algorithms

Research in distributed algorithms consists of the following:

- the identification of the canonical problems that are tractable for research study
- precise statements of those problems
- precise and careful description of algorithms to solve those problems
- rigorous proofs that the algorithms solve the problems
- complexity analysis of the algorithms (time, space, message passing, *etc.*)

- impossibility results

Whereas sequential algorithms are usually easy to reason about and prove, distributed algorithms are often extremely complex. Although the actual code of the algorithm may be very simple, it is the *interleaving* of many processes running the code simultaneously that causes the complexity. Because the interleaving of the steps occurs in some unknown way, it is difficult to understand everything about the possible executions.

To cope with this, one asserts certain *properties* of the execution of the algorithm and then proves these properties in order to prove the algorithm correct.

The interleaving of the execution is not the only factor which makes distributed algorithms more complicated than sequential algorithms. Some others are:

- inherent nondeterminism (*i.e.*, with a given input, there are many ways a computation can unfold, and, in contrast to automata theory, all must be correct.)
- actions at multiple sites (including input and output)
- order of events, rather than just what events occur, may be important (*i.e.*, it is not just a simple function being computed)
- failures
- timing
- continued operation

To handle these and other complications of distributed algorithms, one uses a formal mathematical model of a distributed system. Formal models help mostly in the formulation of precise problem statements. They are also useful in the precise description of algorithms, and are needed for correctness proofs and impossibility proofs. Unfortunately, there are currently a great many different models in existence. Fortunately, there is some attempt at the unification of these models now, with state-machine models becoming more widely accepted.

This course will cover a broad range of topics. Some of the earliest distributed algorithms are *shared-memory* algorithms, and they include many of the basic ideas that will recur elswhere in distributed algorithms. A major area of recent research is in *consensus* algorithms—problems of reaching agreement among a number of distributed processes. These algorithms become interesting when fault tolerance is considered. The formal study of *knowledge* is necessary in order to reason about what information each process has in a distributed system. There are two types of algorithms that deal with computing in a network (other than consensus). *Static network* algorithms deal with networks with fixed inputs that compute some result. *Dynamic network* algorithms have continuous inputs—for example,

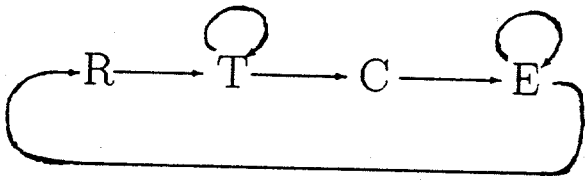


Figure 1.1: The cycle of regions of a single process.

from another algorithm. There is much current work in *concurrency control* algorithms. They implement distributed databases in such a way as to achieve a serial appearance.

It is hoped that by becoming familiar with many of the important distributed algorithms and learning the relevant proof techniques, one will gain an appreciation of the difficulty in designing and proving such algorithms.

1.2 Mutual Exclusion Using Shared Memory

1.2.1 Introduction

Mutual exclusion algorithms are used where n processes $(p_1, \ldots, p_n)$ are contending for a single resource. Each process can be in either the *remainder region* (R), the *trying region* (T), the *critical region* (C), or the *exit region* (E). Normally, a process is in R. When a process is trying to gain control of the resource, it is in T. When it has the resource, it is in C. Eventually, when it is done with the resource, it goes to E and then back to R. The cycle of regions that a process visits is shown in Figure 1.1.

There are two required conditions on mutual exclusion algorithms:

1. *mutual exclusion*—A state in which two processes are in region C is not reachable.
2. *progress*—As long as "operation continues normally", "normal progress" should be made.

"Normal operation" means that a process continues to take steps while in regions T or E and does not take steps while in regions C or R. Furthermore, a process in region C eventually

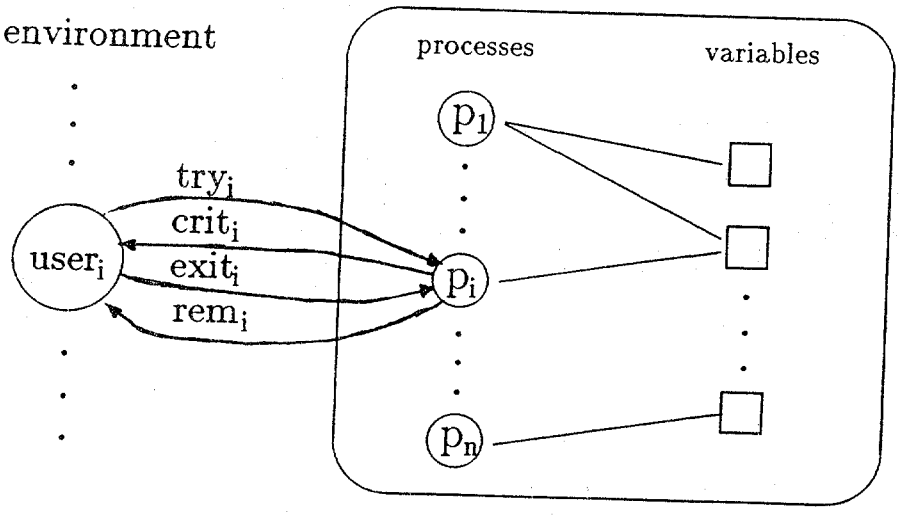


Figure 1.2: A view of the mutual exclusion problem.

leaves (to region E). "Normal progress" means that if some process is not in region R, then some process will eventually change regions. (Note that normal progress does not prohibit lockout—if p_1 enters T, and the rest of the processes are in R, p_1 does not necessarily ever have to go into C. Process p_2 could start cycling through the regions and satisfy normal progress.)

A process p_i can be modelled as a state machine that communicates with the outside world via incoming messages try_i and exit_i, and outgoing messages crit_i and rem_i. One can think of the outside world as an environment of n users ($user_1$, ..., $user_n$). When a $user_i$ wants control of the resource, he sends try_i to p_i. This forces p_i to start the trying code. When p_i has control of the resource (*i.e.*, is in region C), it sends crit_i to $user_i$. When $user_i$ is done with the resource, it sends exit_i to p_i. This forces p_i to start the exit code. When p_i has returned to region R, it sends rem_i to $user_i$.

Note that part of the normal operation requirement is the responsibility of the user rather than the process. The requirement that a process in C eventually leaves means that $user_i$ must send exit_i sometime after receiving crit_i. Assuming that for all i, $user_i$ only sends try_i and exit_i when it should, the algorithm must preserve the cyclic behavior of try_i, crit_i, exit_i, rem_i, ... and must guarantee mutual exclusion.

Each process also has actions involving single *shared variables*. These are shown as squares in Figure 1.2. The two basic actions involving process p_i and a shared variable x are:

1. read x and store its value in a local variable of p_i
2. write a local variable of p_i to x

1.2.2 Dijkstra's algoritm

In 1965, the first mutual exclusion algorithm was developed by Dijkstra. It is presented in Figure 1.3 as code to be run on each process. Label L starts region T. Then, there is a comment where region C would go. The code for region E follows, and finally, a comment where region R would go. There are shared variables: $control[1 \ldots n]$, which can take on values from $\{0, 1, 2\}$, and k, which can take on values from $\{1, \ldots, n\}$. $Control[i]$ is written by p_i and read by all, and k is both written and read by all. The processes are assumed to be asynchronous, and so an *atomic action* must be defined. For this algorithm, atomic actions are reads from and writes to the shared variables. Each atomic action in the code of Figure 1.4 is enclosed in pointy brackets. Note that the read of $control[k]$ does not take two atomic actions because k was just read in the line above, and so a local copy of k can be used. The initial state of the algorithm is where all processes are in region R.

An operational proof

One can show that Dijkstra's algorithm works by direct reasoning about its behavior—an *operational proof.* As mentioned above, many distributed algorithms are too complicated for this approach to be practical, and in those cases an invariant assertional proof is the desired approach. To reason about the algorithm, there must be a concept of *indivisibility* of actions. Above, the basic atomic actions were defined to be reads and writes from and to the shared variables. This will serve as the concept of indivisibility—actions involving shared variables are indivisible, and local computation does not enter into the timing analysis (*i.e.*, local computation happens instantaneously). Note that the algorithm does not clearly specify what is indivisible. Careful reasoning about such algorithms requires removal of these ambiguities.

Theorem 1.1 *Dijkstra's algorithm satisfies mutual exclusion.*

Proof: By contradiction. Assume p_i and p_j were both simultaneously in region C, where $1 \leq i < j \leq n$. Then, $control[i]$ and $control[j]$ were both set to 2 some time before their respective processes entered C. Assume that $control[i]$ was set to 2 first. Then, $control[i]$ remained 2 until p_i left C, which must have occured after p_j entered C. So, after p_j set $control[j]$ to 2, but before p_j entered C, $control[i]$ was always 2. Therefore, p_j must have seen $control[i] = 2$ and so could not have entered C. ∎

Theorem 1.2 *Dijkstra's algorithm satisfies progress.*

Shared variables:

- *control* : an array indexed by [1..n] of integers from {0,1,2}, initially all 0, where *control*[i] is written by p_i and read by all
- k : integer from {1,...,n}, initially arbitrary, where k is written and read by all

Code for p_i

```
**Begin 1st stage, trying to obtain k.**
L: control[i] ← 1
while k ≠ i do
    if control[k] = 0 then k ← i
    end if
end while

**Begin 2nd stage, checking that no other processor has reached this stage.**
control[i] ← 2
for j ∈ {1,...,i−1,i+1,...,n} do **Note order of checks unimportant**
    if control[j] = 2 then goto L
    end if
end for

**Critical region**

control[i] ← 0

**Remainder region**
```

Figure 1.3: Dijkstra's mutual exclusion algorithm.

Shared variables:

- *control* : an array indexed by [1..n] of integers from {0,1,2}, initially all 0, where *control*[i] is written by p_i and read by all
- k : integer from {1,...,n}, initially arbitrary, where k is written and read by all

Code for p_i

```
**Begin 1st stage, trying to obtain k.**
L: S_i ← ∅
⟨control[i] ← 1⟩
while ⟨k ≠ i⟩ do
    if ⟨control[k] = 0⟩ then ⟨k ← i⟩
    end if
end while

**Begin 2nd stage, checking that no other processor has reached this stage.**
⟨control[i] ← 2⟩
for j ∈ {1,...,i−1,i+1,...,n} do **Note order of checks unimportant**
    if ⟨control[j] = 2⟩ then goto L
    end if
    S_i ← S ∪ {j}
end for

**Critical region**
S_i ← ∅
⟨control[i] ← 0⟩

**Remainder region**
```

Figure 1.4: Dijkstra's algorithm showing atomic actions and S_i.

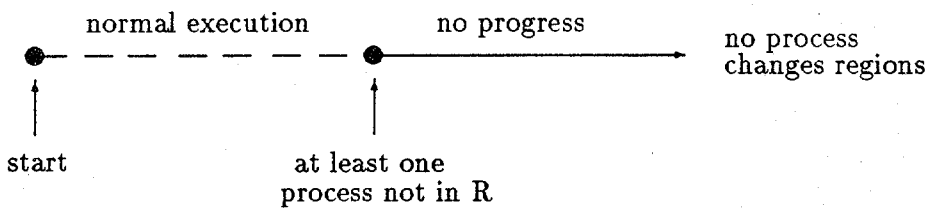


Figure 1.5: No progress.

Proof: Again, by contradiction. There is some execution that reaches a point such that not all processes are in R, and after which no process ever changes region (see Figure 1.5). Note that all processes in T or E continue taking steps. In this state, if one of the processes is in C, then it is guaranteed to reach E, since we assume the environment must send it an exit message. If any of the processes are in E, then after one step they will be in R. Thus, all of the processes are in region T or R, only the processes in region T take steps, no new processes enter T, and all of the processes in T continue to take steps forever.

All contenders (processes in T) keep their $control \geq 1$ during this execution. If k changes during the execution, it is changed to a contender's index. If the value of k starts as a non-contender, then p_i, when it eventually checked, would find that $k \neq i$ and $control[k] = 0$ and thus set k to i. There must exist an i such that this will happen, because all contenders are either in the while loop or in the second stage, destined to fail and return to the while loop. Once k is set to a contenter's index, $control[k] \geq 1$. Then, any future reads of k and $control[k]$ will yield $control[k] \geq 1$, and k will not be changed. So, eventually, k stabilizes to a final (contender's) index.

Once k stabilizes to a contender's index, that contender has little to stand in its way. The while loop is completed and $control[k]$ is set to 2. The only way that p_k could fail to reach C immediately is if $control[i] = 2$, for some $i \neq k$. However, all processes other than p_k whose *control* is 2 eventually return to L, because the assumption is that no processes enter C. Then, they are stuck in stage 1 (because k is not equal to their process id), and can cause p_k no further obstacles. So, eventually p_k enters C. ■

An assertional proof

In general, the assertional technique is more popular than the operational technique of proving distributed algorithms. The idea of an assertional proof is to state *invariants* of the algorithm and then prove by induction that they hold for every reachable state of the system. This usually involves a case analysis of state transitions, but it is often the case that many possible transitions can be easily eliminated (*e.g.*, if they don't affect a variable in question).

To prove mutual exclusion of Dijkstra's algorithm, we must show that

$$\neg[\exists_{i,j} \quad | \quad (i \neq j) \wedge (p_i \text{ in C}) \wedge (p_j \text{ in C})]$$

This alone is not strong enough to prove by induction, however. We must add conditions true of all reachable states. These are the invariants. Let S_i be a local variable of p_i. S_i is a set that contains the indices of all of the processes that p_i encounters in the for-loop whose *control* variable is not 2. Initially $S_i = \emptyset$ for all $1 \leq i \leq n$. When p_i finds $control[j] \neq 2$ (in an iteration of the for-loop), then $S_i \leftarrow S_i \cup \{j\}$. When $S_i = \{1, \ldots, n\} - \{i\}$, the control moves to after the for-loop. When $control[i]$ is set to 0 or 1, $S_i \leftarrow \emptyset$. Figure 1.4 shows where these settings of S_i would appear in the code for p_i.

Let each process have another local variable, a program counter, which keeps track of where in the code that process is. Define the following sets of processes:

- at-for: processes whose program counter is in the for-loop
- before-C: processes whose program counter is right after the for-loop
- in-C: processes whose program counter is in region C

We need to prove that $|\text{in-C}| \leq 1$. However, the stronger statement $|\text{in-C}| + |\text{before-C}| \leq 1$ is sufficient. The following two claims imply that statement:

1. $\neg[\exists_{i,j} \quad | \quad (i \neq j) \wedge (i \in S_j) \wedge (j \in S_i)]$
2. $p_i \in \text{before-C} \cup \text{in-C} \Rightarrow S_i = \{1, \ldots, n\} - \{i\}$

More details of this proof are discussed in Lecture 2.

Lecture 2: September 15

Lecturer: Nancy Lynch *Scribe: Sharon Perl*

2.1 Dijkstra's Mutual Exclusion Algorithm (Cont.)

In Lecture 1 we presented Dijkstra's mutual exclusion algorithm and argued informally that it has the desired safety and liveness properties: it guarantees mutual exclusion and progress. We will begin this lecture by formulating an assertional proof that the algorithm guarantees mutual exclusion.

Before we can prove anything, we must restate the correctness condition more precisely. The following theorem captures our intuitive notion of "guaranteeing mutual exclusion."

Theorem 2.1 *Let s be any state reachable in an execution of Dijkstra's mutual exclusion algorithm. There are no two processes p_i and p_j, $i \neq j$, such that both p_i and p_j are in C (the critical region) in state s.*

We will prove this claim by proving an even stronger claim. Let *in-C* be the set of processes that are in their critical sections, and let *before-C* be the set of processes that are ready to enter their critical sections but have not yet done so (i.e., they are ready to issue crit_i). We will prove the following:

Theorem 2.2 *In any state reachable during an execution of Dijkstra's algorithm* $|\textit{in-C}| + |\textit{before-C}| \leq 1$.

It should be obvious that Theorem 2.1 is a corollary of Theorem 2.2. In order to prove Theorem 2.2 we will need a small set of lemmas and definitions.

Let S_i be a local variable of process i that records the processes already seen to have their control variable set to 2 in stage 2 of the algorithm. S_i will be set to the empty set whenever $control[i]$ is set $\neq 2$. Furthermore, whenever $control[j]$ is examined by p_i in the **for** loop and found to be unequal to 2, then j is added to S_i (see Figure 1.4). S_i is implicit in the state of a process; making it explicit helps us to carry out the proof.

First a small lemma.

Lemma 2.3 *If $S_i \neq \emptyset$ then $control[i] = 2$.*

Proof: From the definition of S_i and the code for the algorithm. ∎

Now, the main lemma that we will use to prove Theorem 2.2 says, in effect, that two processes in stage 2 cannot both miss each other's stage 2 control signals.

Lemma 2.4 $\nexists p_i, p_j\ [i \neq j \text{ and } i \in S_j \text{ and } j \in S_i]$.

Proof: (sketch) By induction on the length of executions. The basis case is easy since, in an execution of length 0, all sets S are empty. Now consider the case where j gets added to S_i (convince yourself that this is actually the only case of interest). This must occur when i is in its **for** loop in stage 2. If j gets added to S_i then it must be the case that $control[j] \neq 2$ because otherwise, i would exit the loop. By the contrapositive of Lemma 2.3 then, $S_j = \emptyset$, and so $i \notin S_j$. ■

The second lemma that we will use to prove Theorem 2.2 implies that when a process is ready to enter its critical region, none of the other processes had its control signal set to 2 when it was last observed by the process.

Lemma 2.5 $p_i \in (before\text{-}C \cup in\text{-}C) \Rightarrow S_i = \{1, \ldots, n\} - \{i\}$.

Proof: By induction on the length of the execution. In the basis case, all processes are in region R, so the claim holds trivially. For the induction hypothesis, assume the claim holds in any reachable state and consider the next step. The steps of interest are those where p_i exits the loop normally (i.e., doesn't **goto** L) or enters C. Clearly, if p_i exits the loop normally, S_i contains all indices except i, since this is the termination condition for the loop (when rewritten explicitly in terms of the S sets). Upon entry to the critical region, S_i does not change. Thus the claim holds in the induction step. ■

We can now prove Theorem 2.2 by contradiction:

Proof: Assume that in some state reachable in an execution $|in\text{-}C| + |before\text{-}C| > 1$. Then there exist two processes, p_i and p_j, $i \neq j$, such that $p_i \in (before\text{-}C \cup in\text{-}C)$ and $p_j \in (before\text{-}C \cup in\text{-}C)$. By Lemma 2.5, $S_i = \{1, \ldots, n\} - \{i\}$ and $S_j = \{1, \ldots, n\} - \{j\}$. But by Lemma 2.4, either $i \notin S_j$ or $j \notin S_i$. This is a contradiction. Thus our assumption must be false and the Theorem must be true. ■

This concludes our examination of Dijkstra's algorithm.

2.2 Comments on Proof Techniques

The term *invariant*, as we will use it in these lectures, is defined to be a predicate true in all reachable states of a program. We use invariants in assertional proofs to prove safety properties of programs.

The proof in the previous section illustrates one form of assertional proof. In it we were able to formulate a series of lemmas asserting invariants about the program and prove each individually, often by induction. This style of proof is nice because each lemma is easy to understand by itself, and the proof of correctness is easy to understand in terms of the lemmas. Sometimes, however, it is necessary to collect the invariants into one large lemma (or theorem) and prove all of them at once. (You might need to do this, for example, if the

truth of invariant 3 for step k is used to prove invariant 5 for step $k+1$.) In such a proof you would first show that all the invariants are true in an empty execution (usually trivial). Then you hypothesize that all invariants are true for executions shorter than some particular length and show that none of the invariants are violated in the next step of the execution.

It is also worth saying a few words about formal techniques for proving progress properties, although we will not carry out the proof for Dijkstra's algorithm (but, see Exercise 5). One common technique involves the use of *variant functions*. A variant function f assigns a measure of progress to each state of the system. The domain of f is the set of states and the range is a *well-founded set* (a set with no infinite decreasing chain). To prove that an algorithm guarantees progress, we must show two things. First, from each reachable state q, the variant function must never increase until some progress is made. Second, in every normal execution from every reachable state q, some step must eventually occur to decrease the value of the variant function, or else progress must occur. Thus, if no progress occurs the variant function decreases, but it cannot decrease indefinitely because its value belongs to a well-founded set. When the variant function finally increases, we are guaranteed to make progress.

A typical example of a well-founded set used to define a variant function is a set of tuples of cardinal numbers, where successive components measure different milestones in the computation (the first component measures the most significant milestone and the last measures the least significant one). Consider a collection of processes that each execute two 5-step tasks. An appropriate value for a variant function might be a two-tuple where the first component counts the number of processes still doing the first task and the second component is the sum of the number of steps remaining for each process in its current task.

2.3 Improved Mutual Exclusion Algorithms

While Dijkstra's algorithm guarantees mutual exclusion and progress, it has a number of undesirable properties as well. For one, it does not guarantee fairness; it is possible that one process will continuously gain access to its critical region while other processes trying to gain access are prevented from doing so. Second, it uses a multi-reader/multi-writer variable (k) which may be difficult or expensive to implement in certain kinds of systems. Finally, it is not resilient to failures of processes. A number algorithms that improve upon Dijkstra's have been designed.

Before we look at improvements to Dijkstra's algorithm, we should first consider what it means for an algorithm to guarantee fairness. Depending upon the context in which the algorithm is used, different notions of fairness may be desirable. Three ideas that have been used are:

1. *eventuality:* an algorithm is fair if eventually all processes trying to enter their critical regions may do so.

2. *time bound:* an algorithm is fair if within some bounded amount of time, any process trying to enter its critical region may do so. (This presupposes some measure of time in the system.)

3. *number of turns waited:* an algorithm is fair if for every process p_i, no process p_j, $i \neq j$, bypasses p_i (goes critical) more than some particular number of times once p_i has entered its trying region.

We will see that different impossibility and complexity results arise depending upon which definition of fairness we use.

In the remainder of this lecture we will look at three additional mutual exclusion algorithms that improve upon Dijkstra's algorithm in different ways, and one impossibility result about mutual exclusion. We will only start to discuss the third algorithm (Lamport's) here, finishing it in the next lecture.

2.3.1 Eisenberg-McGuire Mutual Exclusion Algorithm

The Eisenberg-McGuire Mutual Exclusion Algorithm (see Figure 2.1) guarantees fairness by bounding the number of times that one process can bypass another. It has shared variables named *control*[i] and k, like Dijkstra's algorithm, but uses k somewhat differently. The general idea is that each process, upon leaving C, selects as its successor the next contending process that it discovers by testing the control variables in sequence, storing the successor's index in k. A process in T checks that it is the first known contender starting from k. A process in T also does a last check for k and defers if necessary before entering C. (It is not obvious why this check is needed. See Exercise 2 below.)

The proof that the Eisenberg-McGuire algorithm guarantees mutual exclusion is much like that for Dijkstra's algorithm. The proof of progress goes roughly as follows. Assume that progress is not guaranteed. Then after some point in an execution, only processes in T take steps, and no region changes occur. Thus no process can reach the assignment $k \leftarrow i$, or else it would enter C. Since this is the only statement that modifies k, k does not change. Consider the first contender in the cyclic order $k, k+1, \ldots, n, 1, \ldots, k-1$. We claim that this process meets no resistance: it passes through stage 1 any time it tries, and the other processes in stage 2 eventually drop out as in Dijkstra's algorithm. Thus the first contender will eventually enter C.

To see that the algorithm guarantees fairness, consider any fixed p_i that enters T. From the time p_i enters T until p_i enters C, there is always some process with its control variable set to 1. So each time another process leaves C, it changes k to the next known contender in the cyclic order. That contender must be the next to go; the final test ensures that every other process would defer. So eventually, k is set to i and i goes critical.

Shared variables:

- *control* : an array indexed by [1..n] of integers from {0,1,2}, initially all 0, where *control*[i] is written by p_i and read by all
- k : integer from {1,...,n}, initially arbitrary, where k is written and read by all

Code for p_i

```
**Begin 1st stage, testing if self is first contender after k.**
L: control[i] ← 1
for j = k, k+1, ..., n, 1, ..., k-1 do
    if j = i then exit
    end if
    if control[j] ≥ 1 then goto L
    end if
end for

**Begin 2nd stage, identical to Dijkstra's algorithm.**
control[i] ← 2
for j ∈ {1, ..., i-1, i+1, ..., n} do
    if control[j] = 2 then goto L
    end if
end for

**Begin 3rd stage, making final check.**
if control[k] ≥ 1 and k ≠ i then goto L
end if
k ← i

**Critical region**

**Select as successor the next contending process.**
for j = k+1, ..., n, 1, ..., k-1 do
    if control[j] ≠ 0 then
        k ← j
        exit
    end if
end for
control[i] ← 0

**Remainder region**
```

Figure 2.1: Eisenberg-McGuire Mutual Exclusion Algorithm

2.3.2 Burns' Mutual Exclusion Algorithm

Both of the algorithms we have studied so far use a multi-writer variable (k) along with a collection of single-writer variables (*control*). Because it might be difficult and inefficient to implement (build) multi-writer shared-variables in certain systems (in particular, in distribued systems), algorithms that use only single-writer variables are worth investigating. The next algorithm, developed by Jim Burns, appears in Figure 2.2. Burns' algorithm does not guarantee fairness, but does eliminate the need for a multi-writer variable. Again, the proof that Burns' algorithm guarantees mutual exclusion is similar to the proof for Dijkstra's algorithm, except that the control variable is set to 1 where in Dijkstra's it is set to 2. The proof of progress can be argued by contradiction. Assume that all processes are in either their remainder or trying regions and that they continue taking steps. Partition the processes into those that reach label M and those that do not (call the first set P and the second set Q). Eventually in an execution, we will reach the point where all processes that will ever be in P are in it already. Then we claim that there is at least one process in P (the one among the contenders with the lowest index). Furthermore, we claim that among all the processes in P, the one with the largest index will reach the critical region. (Look at the code and try to convince yourself of this.)

Burns-Lynch Impossibility Result

Burns' algorithm uses no multi-writer variables, but does use n shared (multi-reader) variables to guarantee mutual exclusion and progress for n processes. One might reasonably wonder whether an algorithm that uses fewer than n shared variables could do the same. Certainly, an algorithm with fewer than n single-writer variables would not work, since every process must be able to write something. But what about general read-write systems where every variable can be read or written by any process? The first of a number of impossibility results that we will see in these lectures (this one due to Jim Burns and Nancy Lynch) answers this question negatively.

Theorem 2.6 *If a system S solves mutual exclusion with progress for n processes using read-write shared variables, then S has at least n shared variables.*

The proof of this theorem is complex and we will return to it in a later lecture. The basic idea is to assume that there is an algorithm and construct an incorrect execution, just working from the problem statement (i.e., all we know is that the algorithm guarantees mutual exclusion and progress). A write can be *obliterated* by being overwritten before being read by any other process. In order to go from R to C, a process p must write to some variable that is not *covered* (about to be written) by another process. This is because if it didn't, the rest of the system couldn't distinguish the configuration with p in C from another similar configuration with p in R. But the system must act quite differently in these

Shared variables:

- *control* : an array indexed by [1..n] of integers from {0,1}, initially all 0, where *control*[i] is written by p_i and read by all

Code for p_i

```
L: control[i] ← 0
for j ∈ {1,...,i − 1} do
    if control[j] = 1 then goto L
    end if
end for
control[i] ← 1
for j ∈ {1,...,i − 1} do
    if control[j] = 1 then goto L
    end if
end for
M:
for j ∈ {i + 1,...,n} do
    if control[j] = 1 then goto M
    end if
end for

 **Critical region**

control[i] ← 0

 **Remainder region**
```

Figure 2.2: Burns' Mutual Exclusion Algorithm

two situations: if p is in R, then eventually it must let another process enter C (to ensure progress). However, allowing another process to enter C would yield incorrect behavior from the configuration where p is in C. The proof constructs an involved collection of hypothetical related executions, such that the requirement that processes must write some variable not covered by any other processes actually implies that there must be n separate variables.

A few important points to take away from the brief discussion of this result are:

- Impossibility results in this field arise from the limitations of local knowledge in a distributed system. Every process' actions depend only upon what it sees in its own state and shared memory. Some restrictions (e.g., too few variables) might imply that two executions look the same to some processes even though correctness conditions require them to act quite differently in the two cases.
- n process mutual exclusion with progress requires at least n shared variables (the statement of the result).
- Formal models are extremely important in proofs of impossibility results. Both the algorithm and the correctness conditions must be precisely stated if we're going to have any hope of proving that something is impossible. (For example, if we allow atomic update, the argument used by Burns and Lynch in their proof does not go through.) Some aspects of the model that are important are the specification of "normal operation," and the definition of actions and which components control them.

2.4 Lamport's "Bakery" Mutual Exclusion Algorithm

The last algorithm we will begin to look at in this lecture is Lamport's "Bakery" Mutual Exclusion Algorithm (shown in figure 2.3). This algorithm has a number of desirable properties, in addition to guarantees of mutual exclusion and progress:

1. It guarantees fairness;
2. It uses only single-writer variables;
3. Assuming that failures are detectable, it provides some failure resiliency (failed processes go to their remainder regions and, eventually, their variables get set to 0);
4. It does not require atomic variables.

The primary deficiency of the algorithm is that it requires variables that are unbounded in size (at least theoretically—perhaps in practice bounded size variables would suffice).

An *atomic register* serializes reads and writes in an order consistent with begins and ends of read and write operations. It is very much like the atomic variables that all of the

Shared variables:

- *choosing* : an array indexed by [1..n] of integers from {0,1}, initially all 0, where *choosing*[i] is written by p_i and read by all
- *number* : an array indexed by [1..n] of integers from {0,1,...}, initially all 0, where *number*[i] is written by p_i and read by all

Code for p_i

L1: $choosing[i] \leftarrow 1$
$number[i] \leftarrow 1 + max(number[1], \ldots, number[n])$
$choosing[i] \leftarrow 0$
for $j \in \{1, \ldots, n\}$ **do**
    L2:
    if $choosing[j] = 1$ **then** goto L2;
    end if
    L3:
    if $number[j] \neq 0$ **and** $(number[j], j) < (number[i], i)$ **then** goto L3
    end if
end for

Critical region

$number[i] \leftarrow 0$

Remainder region

Figure 2.3: Lamport's "Bakery" Mutual Exclusion Algorithm

previous algorithms used. Lamport, in his algorithm, uses a register that is weaker than an atomic register. The *safe register* that he uses is has the property that a read that doesn't overlap with any write produces the correct value, while a read that overlaps with a write can produce an arbitrary value.

Lecture 3 discusses the details of Lamport's algorithm.

2.5 Exercises

1. Fill in the details for the assertional proof (outlined in class) of mutual exclusion for Dijkstra's algorithm.

2. What properties are preserved in the Eisenberg-McGuire algorithm if the final check (i.e., the last if-then statement in the trying region) is deleted?

3. Show that lockout is possible in Burns' algorithm.

4. Describe an execution of Lamport's bakery algorithm in which variable values are unbounded.

*5. Give a variant function proof for progress in Dijkstra's algorithm.

*6. Give an assertional proof of mutual exclusion for Lamport's bakery algorithm.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 3: September 20

Lecturer: Nancy Lynch | *Scribe: Stephen Ponzio*

3.1 Lamport's "Bakery" algorithm

The code for this algorithm appears in the notes for Lecture 2.
Lamport's "Bakery" algorithm for mutual exclusion has several attractive properties in addition to satisfying mutual exclusion with progress:

- The shared variables are *single-writer*—only one process may write to a variable. The variables are, however, *unbounded*—their values may grow arbitrarily large.
- It does not require atomic variables, but only *safe registers.*
- It satisfies fairness: the first part of the trying region, the "doorway", is wait-free, and the second part, the "bakery", insures FIFO prioritization.
- It achieves a degree of *failure resiliency.*

3.1.1 Safe Registers

The algorithm does not require *atomic variables.* The access of an atomic variable is modeled in terms of the time when a process requests to read (or write) the variable, t_{req}, and the time when it receives the value of the variable (or an acknowledgement that the value has been written), t_{ack}. Since many processes may be trying to access the variable at the same time, the interval between t_{req} and t_{ack} may overlap for different processes. (Accesses by the same process are assumed to occur serially.) The variables are atomic if for all reads (or writes), the responses are consistent with responses that would be given if each access time could be "shrunk" down to a single point in time, t_{acc}, such that for each access, $t_{req} \leq t_{acc} \leq t_{ack}$ and the entire access could be thought of as happening at t_{acc}. The responses of atomic variables are only required to be consistent with one possible sequence of such "shrunk" points (see Figure 3.1).

The Bakery algorithm requires only *safe registers.* Assuming that writes occur sequentially, the variables must satisfy only one condition: a read that doesn't overlap any writes receives the value of the last write preceding it. A read that overlaps a write may receive any value within the range of that variable. With safe registers, however, it can be very difficult to give assertional proofs because there are many more possible states of the system.

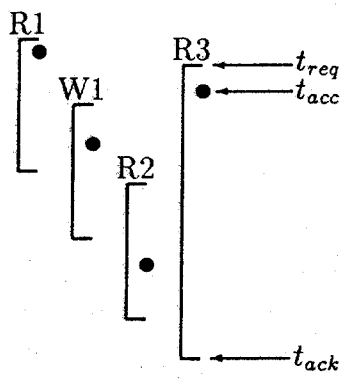


Figure 3.1: Atomic variables: Each bracket represents a read or write cycle, with the vertical dimension representing time. Variables are *atomic* if there exists some set of times t_{acc} such that for each cycle's t_{acc}, the entire cycle could be conceptually "shrunk" to that point in time without changing the values of any responses.

3.1.2 Fairness

The trying region $\mathcal{T}$ is broken up into two subregions: $\mathcal{T}_1$ and $\mathcal{T}_2$. $\mathcal{T}_1$, the "doorway", consists of the beginning of setting *choosing* to 1 to the end of setting *choosing* to 0. In the doorway, a process chooses a value *number* which is the greater than the highest *number* already chosen by processes executing the algorithm. While it does this, it sets *choosing*$[i]$ to 1 to let other processes know that it is currently choosing a value. $\mathcal{T}_2$ is the **for** loop: a process checks to see if its *number* is the lowest, waiting for any processes that are *choosing*. The "bakery" consists of $\mathcal{T}_2$ together with the critical region $\mathcal{C}$. The algorithm resembles the operation of a bakery: customers enter the doorway, choose a number, exit the doorway, and wait in the store.

Passage through the doorway is guaranteed in a fixed number of steps. There is no loop in $\mathcal{T}_1$; a process must only compute the maximum of all other processes *number* variables. (Of course, a slow process may be passed in the doorway by a faster process.)

Once a process p_i has entered $\mathcal{T}_2$, it is guaranteed to advance to $\mathcal{C}$ (critical region) ahead of any other process that later enters $\mathcal{T}$. Since p_i has already chosen *number*$[i]$, a process that enters $\mathcal{T}$ later must choose a *number* that is greater.

Claim 3.1 *If p_i is in $\mathcal{C}$ and p_j is in the bakery, $i \neq j$, then $(number[i], i) < (number[j], j)$ (where the pairs are ordered lexicographically).*

Proof: In loop L2, p_i reads *choosing*$[j] = 0$. Thus, either p_j is not in the $\mathcal{T}_1$ or p_j is just setting *choosing*$[j]$ (either $0 \leftarrow 1$ or $1 \leftarrow 0$).

Case 1. p_i reads *choosing*$[j]$ either totally before p_j's enters $\mathcal{T}_1$ or while p_j is entering $\mathcal{T}_1$ (i.e., setting *choosing*$[j] \leftarrow 1$).

In this case, p_i chooses *number*$[i]$ before p_j starts max, and thus *number*$[i] <$ *number*$[j]$.

Case 2. p_i reads *choosing*$[j]$ totally after p_j exits $\mathcal{T}_1$ or while p_j is exiting $\mathcal{T}_1$ (i.e., setting *choosing*$[j] \leftarrow 0$).

In this case, p_i reads the correct value of *number*$[j]$ in L3. Since p_i is able to exit $\mathcal{T}_2$, we must have $(number[i], i) < (number[j], j)$.

■

Corollary 3.2 *The bakery algorithm satisfies mutual exclusion.*

Proof: By contradiction: if two processes p_i and p_j are both in $\mathcal{C}$, then by Claim 3.1, we have $(number[i], i) < (number[j], j)$ and $(number[i], i) > (number[j], j)$ simultaneously, which is impossible. ■

3.1.3 Progress

Progress under normal conditions is fairly easy to verify: without waiting, all processes may enter $\mathcal{T}_2$, at which point the process with the lowest pair $(number[i], i)$ may proceed unhindered.

The Bakery algorithm also guarantees progress in the presence of certain types of failures. Allowable failures are those in which a process goes back to $\mathcal{R}$ (remainder region) and eventually has all of its registers set back to 0. A process is only allowed to fail once; it may not be restarted.

Theorem 3.3 *During a normal execution in which failures of the specified kind are allowed, if there is some process not in $\mathcal{R}$ that never fails, then some process eventually reaches $\mathcal{C}$ or $\mathcal{R}$ via a* non-failing *transition.*

Proof: Restrict attention to that part of the execution where all processes that are going to fail have already failed. Now the algorithm continues as before, with fewer processes. A failed process cannot cause a non-failed process to get stuck in L2 since a failed process must eventually have *choosing* set back to 0. Likewise, it must have *number* set back to 0 and thus no non-failed process can get stuck in L3. ■

Note that if a failed process were allowed to restart, it would be possible for it to lock out another process by continually setting *choosing* to 1 before the non-failed process could exit L2.

Shared variables:

- q : an array indexed by {0,1} of values from {*nil*,T=1,F=0}, initially all *nil*, where $q[i]$ is written by p_i and read by all

Notation: $opp(i) = \neg i$ **the opponent of i**

Code for p_i

$q[i] \leftarrow$ **if** $q[opp(i)] = nil$ **then** T **else** $i \oplus q[opp(i)]$

$q[i] \leftarrow$ **if** $q[opp(i)] = nil$ **then** $q[i]$ **else** $i \oplus q[opp(i)]$

wait until $q[opp(i)] = nil$ **or** $(i \oplus (q[opp(i)] \neq q[i]))$

Critical region

$q[i] \leftarrow nil$

Remainder region

Figure 3.2: Peterson's 2-process mutual exclusion algorithm

3.2 Peterson-Fischer 2-Process algorithm

In addition to satisfying mutual exclusion with progress, this 2-process algorithm also uses single-writer variables of bounded size. The algorithm (shown in Figure 3.2) also satisfies the same failure-tolerance conditions as the Bakery algorithm even when failed processes are allowed to restart after they have entered $\mathcal{R}$ and had their variables reset.

3.2.1 Operation

The algorithm works fairly simply—only p_0 and p_1 compete. Upon entering $\mathcal{T}$ and checking to see if the other process is also in $\mathcal{T}$ or $\mathcal{C}$, p_0 sets $q[0] \leftarrow q[1]$ which satisfies p_1's wait condition. p_1 sets $q[1] \leftarrow \neg q[0]$ which satisfies p_0's wait condition. The two conditions obviously cannot be satisfied simultaneously. The wait condition for either process is satisfied if the other process is in $\mathcal{R}$ ($q[opp(i)] = nil$).

To see why we need two tests of whether the opponent is in $\mathcal{T}$ or $\mathcal{C}$, consider the following scenario using only one test:

time ↓	**steps of p_0**	**steps of p_1**
	reads $q[1] = nil$	
		reads $q[0] = nil$
	sets $q[0] = \mathrm{T}$	
	reads $q[1] = nil$	
	enters $\mathcal{C}$	
		sets $q[1] = \mathrm{T}$
		reads $q[0] = \mathrm{T}$
		sees $1 \oplus (\mathrm{T} \neq \mathrm{T})$ is satisfied
		enters $\mathcal{C}$

With the second test, it is impossible for both processes to enter the **wait** loop without recognizing that the other process is also in $\mathcal{T}$ or $\mathcal{C}$.

3.2.2 Fairness

The algorithm satisfies fairness according to the following no-lockout condition:

Theorem 3.4 *During normal execution in which failures of the specified kind are allowed, if some process p is outside $\mathcal{R}$ and doesn't fail, then p eventually makes a non-failing transition to $\mathcal{C}$ or $\mathcal{R}$.*

Proof: When p_i enters the **wait** loop, if $q[opp(i)] = nil$ then p_i may enter $\mathcal{C}$.
If $q[opp(i)] \neq nil$ then either

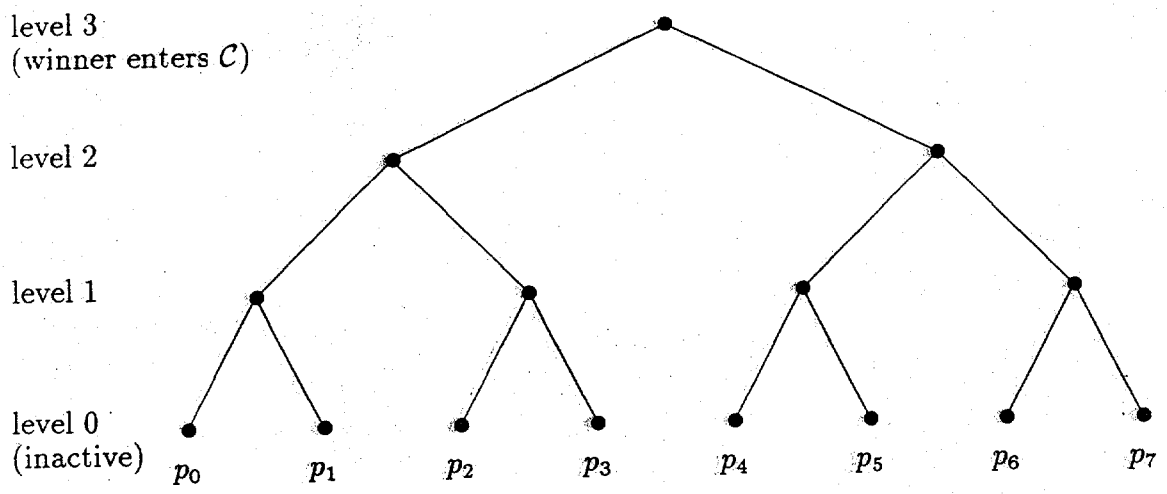


Figure 3.3: The playoff tree for the Peterson-Fischer n-process algorithm on 8 processes.

$$q[opp(i)] = q[i] \quad \text{or} \quad q[opp(i)] \neq q[i]$$

and the wait condition is satisfied for either p_i or $opp(p_i)$. If the wait condition is satisfied for $opp(p_i)$, then $opp(p_i)$ will advance to $\mathcal{C}$ and eventually leave $\mathcal{C}$. Once $opp(p_i)$ has entered $\mathcal{C}$, it can only cause p_i's wait condition to become true: upon leaving $\mathcal{C}$, it will set $q[opp(p_i)] \leftarrow nil$, and if it reenters $\mathcal{T}$, it will see that p_i is waiting and defer to it.[1] ■

3.3 Peterson-Fischer n-Process algorithm

The Peterson-Fischer 2-Process algorithm is generalizable to n processes by considering competition for $\mathcal{C}$ as competition in a tournament (see Figure 3.3). At each level, the 2-process algorithm is run and the winner advances to the next level until finally the winner of the top level is allowed to enter $\mathcal{C}$.

As p_i advances up the playoff tree, it plays the role of p_0 if it comes up the left branch and p_1 if it comes up the right branch. Thus, the possible roles of a process are known ahead of time, given by the function bit(i, k) (see Figure 3.4).

Also predetermined is the set of potential opponents for a process at a given level, denoted by opponents(i, k) (see Figure 3.4). Finally, each process has an pair $(level, flag)$ associated with it, where *level* is the level of the tree at which the process is competing (0 meaning that a process is not in $\mathcal{T}$), and *flag* taking on the function of $q[i]$ in the 2-process algorithm.

When a process p_i is ready to compete at level k, it uses the subroutine $OPP(i, k)$ to find its opponent, if any are ready to compete at level k or above. If the process p has no

[1] Some of these proof sketches are rather informal and are not a substitute for careful formal proofs.

opponent or it beats its opponent, it advances to the next round. Otherwise, if its wait condition is not satisfied or its opponent has already advanced above that level, the process waits for its wait condition to become true.

3.3.1 Mutual exclusion

Theorem 3.5 *The n-process algorithm satisfies mutual exclusion.*

Proof: The main idea is that no two processes can be past a common opponent at the same time—no two processes can have a common "ancestor" which they both have passed.

By contradiction: assume an execution in which both p_i and p_j are in $\mathcal{C}$ at once. Then they both must have passed a common ancestor. Let k be the lowest level at which two processes pass a common ancestor. Call these processes p_r and p_s. Assume without loss of generality that p_r had its wait condition satisfied first. At that time, p_r either saw p_s below level k or saw p_s at level k and beat it. Eventually thereafter, p_s reaches level k and discovers p_r as its opponent. Now, as long as p_s continues to identify p_r as its opponent and p_r remains at a higher level or in $\mathcal{C}$, p_s must wait. Thus, then the only way p_s can advance past k is to identify another opponent p_t. But this means that both p_r and p_t advanced to level k, implying that k was *not* the first level at which two processes have a common ancestor: p_r and p_t came up the same branch to get to level k, and thus must have already have passed a common ancestor at level $k-1$. This contradicts the choice of k as the lowest such level. Thus, there can be no level k and mutual exclusion is satisfied. ∎

3.3.2 Progress and no lockout

Assume a normal execution in which some process remains in $\mathcal{T}$ forever. Let $\mathcal{P}$ be the set of processes that get stuck in $\mathcal{T}$. Each process in $\mathcal{P}$ eventually reaches some final level at which it gets stuck; let p_i be stuck at the highest such level. Then p_i waits for a process p_j which is either higher than p_i or at the same level with conditions favorable to p_j. In the first case, p_j is not one of the stuck processes since p_i is the highest stuck process and p_j is higher. In the second case, p_j can advance and again is not one of the stuck processes. In either case, p_j eventually enters $\mathcal{C}$ and subsequently $\mathcal{R}$. No further opponents can bypass p_i at level k (as for the 2-process algorithm) and p_i will advance. This contradicts the assumption that p_i is stuck; thus, no process can be locked out.

However, it is possible for a slow process to get passed an arbitrary number of times before making progress. Consider a process p_i at level $k < \log n$. Suppose the wait condition for p_i is satisfied, but p_i is slow to advance. Another process p_j from the "other half" of the tree may reach the top level before p_i, and seeing no opponent, enter $\mathcal{C}$. If p_j is fast enough, it could reenter the protocol and again advance to the top level before p_i. It is possible for this to happen an aribtrary number of times.

Shared variables:

- q : an array indexed by $\{0, 1, \ldots, n\}$ of pairs (level,flag), where level is an integer and flag takes on values in {T,F}. Initially, $q[i]$ = (0,F) for all i. Variable $q[i]$ is written by p_i and read by all.

Notation:

- The function bit(i, k) tells what role p_i plays in level k competition; roles obtainable from binary representation. That is, bit(i, k) = bit number $(\log n - k + 1)$ of the binary representation of i.
- Let opponents(i, k) denote all potential opponents for p_i at level k.

Subroutine *OPP*(i, k): (Purpose: to search for opponent.)

```
for j ∈ opponents(i,k) do
  opp ← q[j]
  if level(opp)≥ k then return(opp)
return(0,F)
```

Code for p_i

```
for k = 1,...,log n do
  opp← OPP(i,k)
  q[i] ← if level(opp) = k then (k,bit(i,k) ⊕ flag(opp)) else (k,T)
  opp← OPP(i,k)
  q[i] ← if level(opp) = k then (k,bit(i,k) ⊕ flag(opp)) else q[i]
  L: opp← OPP(i,k)
  if (level(opp) = k and (bit(i,k) ⊕ (flag(opp) = flag(q[i])))) or level(opp) > k then
    goto L
**Critical region**

q[i] ← (0,F)

**Remainder region**
```

Figure 3.4: Peterson's n-process mutual exclusion algorithm.

6.852 Distributed Algorithms **Fall Semester, 1988**

Lecture 4: September 22

Lecturer: Nancy Lynch *Scribe: Azer Bestavros*

4.1 Peterson-Fischer n-Process Algorithm (Cont.)

In Lecture 3 we presented Peterson-Fischer's tournament algorithm and argued that it satisfies some desired properties, namely: mutual exclusion, progress and no lockout. We now turn to another important aspect that we overlooked in our previous discussion of mutual exclusion algorithms, namely *complexity analysis.*

4.1.1 Space Analysis

Here we are interested in the number of variables and the size of these variables. For n processes, Peterson-Fischer's tournament algorithm uses n variables. Each of these variables might assume one out of $2 \log n$ values[1]. Thus $\mathcal{O}(n \log n)$ values are needed.

4.1.2 Time Analysis

For asynchronous algorithms, it is not obvious how time complexity should be measured. For instance, we can't just count the number of steps as with Turing Machines because of the uncertainty associated with "busy-waiting" steps. Also, it is very restrictive to assume that each step takes a fixed amount of time since this would result in limiting the possible interleavings and thus would result in time measures that work only for some possible executions.

A better approach would be to assume upper bounds on the time required for each step and use these to infer upper bounds on higher-level events. For instance we might assume that all steps take time in the interval $(0, a]$ for some constant a, and then infer an upper bound on the time a process has to wait in the trying region in order to enter the critical region[2].

- For the Tournament algorithm, let a be the upper bound on the step time. We also need an upper bound on the time inside the critical region[3]. Let b be that bound.

[1] That is an $\mathcal{O}(loglog(n))$ bits is required per variable.

[2] The constant a can be different for different processes – this would complicate the analysis.

[3] Otherwise, a process could stay in the critical region for any amount of time and block other processes, which would prevent deriving any upper bounds.

- Let $T(k)$ be the maximum time it takes a process[4] to enter $\mathcal{C}$ after winning at level k , where $1 \leq k \leq \log n$. Solving for $T(0)$ gives the required upper bound since that represents the time from when a process enters $\mathcal{T}$ to when it enters $\mathcal{C}$. Moreover, $T(\log n) \leq a$, since only one step is needed to enter $\mathcal{C}$ after winning at the top level. Now, in order to find $T(0)$, we have to find a recurrence relation for $T(k)$ in terms of $T(k+1)$.

- Suppose that p_i has just won at level k and advances to level $k+1$. Now, p_i has to call the OPP subroutine. The maximum time it takes to complete the $OPP(i, k+1)$ call is $\mathcal{O}(a2^k)$ since for a tree of depth $k+1$, p_i has to check 2^k leaves. Thereafter, p_i has to perform some assignments[5] before it reaches the wait loop, for which two cases can be singled out:

  1. P_i finds the condition to be true the first time it tests it. Hence, p_i immediately wins at level $k+1$ thus we have:

$$T(k) = \mathcal{O}(2^k) + T(k+1)$$

  2. P_i finds the condition to be false the first time it tests it. This means that there is a competitor (say p_j) at a level $\geq k+1$. P_i will have to wait for at most $\mathcal{O}(2^k)$, since by that time p_j must have reached its wait loop. Now, either p_i or p_j should be a winner.

     (a) If p_i is the winner, it need not wait so we get

$$T(k) = \mathcal{O}(2^k) + T(k+1)$$

     (b) If p_j is the winner, then it needs a time $\mathcal{O}(2^k) + T(k+1)$ to get to $\mathcal{C}$, and a maximum of b to get out of $\mathcal{C}$. Thereafter, nothing can be blocking p_i. Thus,

$$T(k) = \mathcal{O}(2^k) + 2T(k+1)$$

- Taking the worst case, we end up with the following recurrence:

$$T(k) \leq \mathcal{O}(2^k) + 2T(k+1), \text{ where}$$
$$T(\log n) \leq a$$

- Solving the above recurrence, we get a solution for $T(0)$ which is the required bound on the time from when a process enters $\mathcal{T}$ to when it enters $\mathcal{C}$.

$$T(0) \leq \mathcal{O}(n^2)$$

[4] We assume the process won't fail.

[5] This requires some maximum constant time that we ignore.

Shared variable v: a single shared variable taking on values from $\{0, 1\}$, initially 0.

Code for p_i:

```
waitfor v = 0
v ← 1
**Critical region**
v ← 0
**Remainder region**
```

Figure 4.1: A trivial test-and-set algorithm for achieving mutual exclusion

4.2 Test-and-Set Algorithms

We have concluded our discussion of read/write shared memory algorithms, and now we move to another shared memory model called *test-and-set*. It is interesting to note how a small change in the model makes a drastic difference in the kinds of impossibility results one can get. Test-and-set algorithms assume the existence of a strong primitive that enables an atomic *read-compute-write* access to shared variables. This assumes that some low-level arbitration mechanism manages such requests. Paradoxically, assuming the availability of a mechanism that guarantees fair exclusive use of shared memory does not directly guarantee fair exclusive use of the critical region.

4.2.1 A simple Test-and-set Algorithm for mutual-exclusion

The test-and-set algorithm shown in Figure 4.1 guarantees mutual-exclusion and progress. In this algorithm, the variable v is used like a semaphore. It is assumed that the variable is tested continuously until a 0 is found, then without releasing the variable (atomically), it is set to 1, then released. The algorithm is unfair since lockout is possible.

In describing test-and-set algorithms, one has to be careful about the indivisibility of actions. To be explicit, we introduce the special purpose constructs *lock* and *unlock* into the language to mark the beginning and end of the exclusive access to the shared variables.

Shared variable v: a single shared variable taking on values from $\{0, 1\}$, initially 0.

Code for p_i:

waitfor $v = 0$
$v \leftarrow 1$
unlock
Critical region
lock
$v \leftarrow 0$
unlock
Remainder region
lock

Figure 4.2: A trivial test-and-set algorithm (rewritten)

Moreover, we redefine the *waitfor* C construct as follows:

while $\neg C$ **do** unlock ; lock

Figure 4.2 shows the same algorithm given above, now rewritten using these new constructs. Note that a process always enters the trying region with the variable locked.

Notice how a small change in the underlying model makes a drastic changes in the results. For instance, using the R/W model, at least n variables were needed, whereas the above algorithm uses only 1 variable and only 2 values.

4.2.2 Fair mutual-exclusion using test-and-set algorithms

In the simple algorithm given above, lockout is possible. However, we can get fairness back by forcing a FIFO behavior. This can be done by substituting the variable v with a queue. Now, at the beginning of $\mathcal{T}$, a process adds itself to the queue (atomically), and waits for its turn to enter $\mathcal{C}$. On the other hand, when leaving $\mathcal{C}$, a process removes itself from the queue, thus enabling the next process to enter $\mathcal{C}$. The algorithm is simple and fast, but

space-consuming. It uses up to n shared variables, each having up to n values. Thus $\mathcal{O}(n^n)$ values are needed.

One way of reducing the space requirement in the above solution is to have each process grab a ticket as it enters $\mathcal{T}$ and then waits for its turn to enter $\mathcal{C}$. This solution requires the use a shared variable where the next ticket to be issued, *next_ticket*, and the ticket permitted to enter $\mathcal{C}$, *permitted_ticket*, are stored. At the beginning of $\mathcal{T}$, a process will have to read the shared variable to get and increment the *next_ticket* field. Again, this read-compute-write is assumed to be done atomically. When, a process gets out of $\mathcal{C}$, it will have to increment the *permitted_ticket* field of the shared variable. All computations are done in $mod(n)$, thus each of the two fields can assume n different values making the space requirement $\mathcal{O}(n^2)$. The following theorem states that n is, as a matter of fact, a lower bound on the number of values needed to achive mutual-exclusion, progress, and bounded waiting[6].

Theorem 4.1 *Let $\mathcal{S}$ be a system of n processes, $n \geq 1$, and q be any configuration. Assume $\mathcal{S}$ satisfies mutual exclusion, progress, and bounded waiting. Then, the shared variables of $\mathcal{S}$ can take on at least n values.*

Proof:

- Let $V(q)$ be the value of the shared variables in q.
- We say that q *looks like* q' to a process p_i if:
  1. $V(q) = V(q')$, and
  2. The state of p_i is the same in q and q'.
- For a schedule h, we define $result(q, h)$ to be the configuration that results from applying the schedule h starting from configuration q.
- Let $exit(q)$ be a schedule that would result in having all processes in $\mathcal{R}$, where only the processes not in $\mathcal{R}$ in q are allowed to appear in the schedule. Note that by the normal operation assumption and the fact that $\mathcal{S}$ guarantees progress this schedule is always possible.
- We define $enter(q, i)$ to be the schedule that contains steps taken only by process p_i, starting from a configuration q in which all processes are in $\mathcal{R}$. Thus, p_i will eventually end up in $\mathcal{C}$.
- Let $q_0 = result(q, exit(q))$. In q_0 all processes are in $\mathcal{R}$.

[6] no claims about how many times a process in $\mathcal{T}$ will be bypassed.

- Let $q_1 = result(q_0, enter(q_0, 1))$. In q_1, process p_1 is in $\mathcal{C}$, whereas all other processes are still in $\mathcal{R}$.
- Now, let each process p_j, $2 \leq j \leq n$, enter $\mathcal{T}$ in turn. This can be done by the following sequence of configurations: $q_j = result(q_{j-1}, j)$, where $2 \leq j \leq n$.
- We claim that $V(q_i) \neq V(q_j)$, for $0 < i < j \leq n$. We prove this claim by contradiction as follows:
  - Assume that for some i and j, where $0 < i < j \leq n$, we have $V(q_i) = V(q_j)$. It follows that q_i looks like q_j to processes $p_1, p_2, \ldots, p_i$.
  - Starting from q_i, there exists a schedule h that yields a normal execution and which involves only $p_1, p_2, \ldots, p_i$ and results in some process to enter $\mathcal{C}$ infinitely many times.
  - The same schedule h when run from q_j will make $p_1, p_2, \ldots, p_i$ run exactly the same as before while $p_{i+1}, p_{i+2}, \ldots, p_n$ will always remain in $\mathcal{T}$. Note that, starting from q_j, this is not a normal execution, since $p_{i+1}, p_{i+2}, \ldots, p_n$ are not taking steps in h – otherwise they should eventually enter $\mathcal{C}$. However, by running a sufficiently long, but *finite* prefix of h from q_j, it will be always possible to violate the bounded waiting assumption of $\mathcal{S}$ which contradicts the basic assumption that $\mathcal{S}$ achieves bounded waiting.
- Hence $V(q_i) \neq V(q_j)$, for $0 < i < j \leq n$. Thus, we need at least n values for the shared variables used in $\mathcal{S}$.

■

In Lecture 5 we will present a deterministic algorithm that cuts down the number of values to $\mathcal{O}(n)$.

4.3 Exercises

1. Does Burns' mutual exclusion algorithm work if the shared registers are all safe registers? Why or why not?

2. Explain why both of the first two assignment statements are necessary in the Peterson-Fischer 2-process mutual exclusion algorithm.

3. Prove an upper bound on the time required for a particular process to reach its critical region, from the time when it enters its trying region, in the Lamport bakery algorithm. (Assume for simplicity that the shared registers are atomic rather than just safe.)

4. Write the code for a simplified version of the Burns, Fischer, et al bounded-waiting test-and-set mutual exclusion algorithm that allows a permanent "supervisor" process.

*5. (*Open*) Give an (elegant) assertional proof for the Peteson-Fischer 2-process algorithm.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 5: September 27

Lecturer: Nancy Lynch | *Scribe: Jon Riecke*

5.1 More on the Test-and-Set Model

5.1.1 Burns, et al. Mutual Exclusion Algorithm

Lecture 4 introduced the *test-and-set* model. In this model, a process may obtain exclusive access to a shared variable through the use of *lock* and *unlock* actions. A fair algorithm has been designed for this model with the additional property of *bounded waiting*, the Burns-Fischer-Jackson-Lynch-Peterson algorithm. In this algorithm, a single variable is shared and only stores $n + c$ different values. Since any test-and-set, bounded-waiting algorithm requires at least n values (proven in Lecture 4), this algorithm's storage requirements are quite close to optimal.

The best way to think of the algorithm is to imagine a special "supervisor" process who coordinates the other processes. It is up to the supervisor to guarantee that no two processes enter the critical region at the same time, and that no process waits an unbounded or infinite time to get to the critical region.

The algorithm works in three phases. Processes that enter the trying region accumulate in a "buffer," where their relative order is lost. When the supervisor gains control of the shared variable, the supervisor moves all buffered processes into a "main" section, and finally dispatches them, one at a time, to their critical regions (see Figure 5.1). Once entering "main," a process is assured of reaching its critical region ahead of any process who later enters the buffer. Thus, process are bypassed at most once, and so the algorithm meets the condition of bounded-waiting.

The notion of the "buffer" and "main" sections are only analogies. In reality, the supervisor will not maintain a list of processes in each section, but counters *buff* and *main* of how many processes are in each section. The processes themselves will know, by their program counters, into which sections they fall.

The supervisor communicates with the processes via a shared variable v. The supervisor may send the messages ENTER and ELECT to the processes: ELECT tells a process to move from the "buffer" to the "main" sections, and ENTER tells a process to proceed from the "main" section to its critical region. Of course, any process waiting for a message from the supervisor may get the message the supervisor sends. The processes themselves send ACK messages acknowledging the receipt of the ENTER and ELECT messages, and BYE messages when they leave their critical regions.

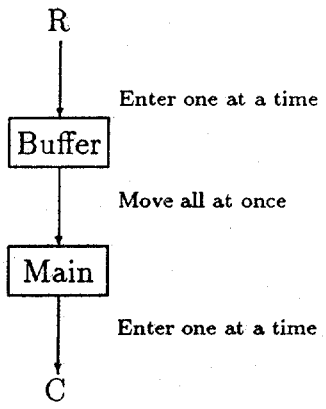


Figure 5.1: A conceptual view of the Burns, *et al.* algorithm for mutual exclusion.

Processes also need to tell the supervisor when they move into their trying regions. A single message will not work here; there is no way to guarantee that the supervisor will be the next process to lock v. So instead of using a single message, v will store a count also. When the supervisor gains control of the variable, v will tell how many processes have entered their trying regions since the last check. The supervisor then adds the count to its local variable *buff*, resets the count in v to be 0, and continues.

How many values can v store? It must store at most one message and a counter; if there are c messages (here, $c = 4$), v could take on $c \cdot n$ different values. The solution is still a bit far from the $n + c$ values promised.

We need a trick. Suppose v may store a count *or* a message; then v will have $n + c$ possible values. To make the algorithm work, if a process p_i needs to update a count and the variable v contains a message, p_i will "steal" the message, write a 1 to v, and wait for v to be reset to 0. If p_i ever reads v and sees a 0, it can return the stolen message to v. To see that p_i must see a 0 in v in a bounded amount of time, note that at the time p_i steals the message, the supervisor can be doing one of three things:

1. Moving processes from the "buffer" to the "main" sections;

2. Moving a process from the "main" section to the critical region; or

3. Waiting for a process to finish with the critical region.

If the supervisor is moving processes from the "buffer" to the "main" sections, the system will eventually quiesce (*i.e.*, no messages will be sent) when all processes but one have been moved into the "main" section. Then v will stay 0 until p_i reads it; p_i can then return the

Values for shared variable v: ENTER, ELECT, ACK, BYE

Notation: Let $\mathcal{N}$ denote the set $\{0, 1, \ldots\}$.

Process Protocol: With local variable m

```
if v ∈ N then v ← v + 1
   else [m ← v; v ← 1; waitfor v = 0; v ← m;]
waitfor v = ENTER; v ← ACK;          ** Buffer **
waitfor v = ELECT; v ← ACK;          ** Main **
v ← 0
unlock;

 ** Critical Section **

waitfor v = 0; v ← BYE;

 ** Remainder Region **
```

Supervisor Protocol: With local variables *main, buff*

```
L: if main = 0 then    ** Move processes from buffer to main **
      while buff + v > 0 do
         buff ←buff +v − 1; v ← ENTER; main ←main + 1;
         while v ≠ ACK do
            if v ∈ N then
               buff ←buff +v; v ← 0;
            unlock; lock;
         v ← 0;

  ** Move processes, one at a time, to critical region **

while main > 0 do
   main ←main −1; v ← ELECT;
   while v ≠ ACK do
      if v ∈ N then
         buff ←buff +v; v ← 0;
      unlock; lock;
while v ≠ BYE do
   if v ∈ N then [buff ←buff +v; v ← 0;]
   unlock; lock;
goto L;
```

Figure 5.2: The Burns *et al.* test-and-set mutual exclusion algorithm in simplified form.

Values for shared variable v: FREE, ENTER, ELECT, ACK, COUNT, BYE

Local variables: *m, main, buff*

Trying protocol:

```
if v = FREE then v ← 0
  else
    if v ∈ 𝒩 then v ← v + 1
      else [m ← v; v ← 1; waitfor v = 0; v ← m; m ← 0;]
    waitfor v = ENTER; v ← ACK;          ** Buffer **
    waitfor v = ELECT; v ← ACK;          ** Main **
    while v ≠ BYE do          ** Receive Counts **
      if v ∈ 𝒩 then [buff ←buff +v; v ← 0;]
      if v = COUNT then [main ←main +1; v ← ACK;]
      unlock; lock;
    v ← 0
unlock;

 ** Critical Section **

lock;
```

Figure 5.3: Burns, *et al.* Mutual Exclusion Algorithm, trying protocol.

stolen message to v. A similar quiescent state will occur in the other two cases. Figure 5.2 gives code for this simplified description of the algorithm.

The algorithm requires yet more refinement, since one process is designated as a supervisor and cannot do anything else. The idea is a common one in the distributed algorithms literature: distribute the job of "supervisor" among the processes. In this algorithm, a process becomes the supervisor when it is next to enter the critical region. In the exit protocol, the process then passes the job of "supervisor" on to another process and sends it all the necessary state information. The algorithm requires a two more messages, FREE and COUNT. The complete algorithm is given in Figures 5.3 and 5.4.

5.1.2 Lower Bounds for No Lockout

The proof that test-and-set mutual exclusion requires n values relied upon bounded waiting; in the no-lockout case, we might need far fewer than n values. Another impossibility result shows that any algorithm still needs $O(n)$ values, although the actual number is roughly

Exit Protocol:

```
if main = buff = v = 0 then v ← FREE
   else
      if main = 0 then        ** Move processes from buffer to main **
         while buff + v > 0 do
            buff ←buff +v − 1; v ← ENTER; main ←main + 1;
            while v ≠ ACK do
               if v ∈ 𝒩 then
                  buff ←buff +v; v ← 0;
               unlock; lock;
            v ← 0;

 ** Elect new supervisor **

      buff ←buff +v; v ← ELECT; main ←main −1;
      while v ≠ ACK do
         if v ∈ 𝒩 then
            buff ←buff +v; v ← 0;
         unlock; lock;

 ** Send counts **

      while main > 0 do
         v ← COUNT; main ←main −1; waitfor v = ACK;
      v ←buff;
      waitfor v = 0; v ← BYE;
unlock;

 ** Remainder region **

lock;
```

Figure 5.4: Burns, *et al.* Mutual Exclusion Algorithm, exit protocol

$\frac{n}{2}$. There is a somewhat technical restriction in this theorem that does not quite cover all no-lockout test-and-set algorithms; the algorithms not considered, though, are few. (There is an $\Omega(\sqrt{x})$ lower bound in general, a result due to Burns, Fischer, Jackson, Lynch, and Peterson.)

An algorithm using $\frac{n}{2} + c$ values can also be found for this case, using much the same idea as in the Burns, *et al.* algorithm. A supervisor puts processes to sleep when there are more than $\frac{n}{2}$ waiting, and wakes them up when the supervisor goes to the critical region.

5.2 k-Exclusion

A slightly different problem, k-exclusion, may also be posed in the distributed setting. A protocol solves the k-exclusion problem if no more than k processes can be in their critical regions simultaneously. The progress condition must be modified slightly: if at most $k-1$ processes stop in the critical region, then progress should still be made.

Of course, any mutual exclusion algorithm solves the k-exclusion problem, albeit inefficiently using the shared resource. A number of algorithms have appeared in the literature to solve the problem more efficiently. For example, one algorithm uses a colored ticket scheme, using only $O(n^2)$ values (the constant depending on k.) See the bibliography for papers containing these algorithms [FischerLBB85a,FischerLBB85b,DolevGS88].

5.3 Atomic Mutual Exclusion Requires n-Variables

We have seen that in the test-and-set model, any bounded-waiting algorithm requires n values. For the atomic register case, the situation is quite different. A remarkable theorem due to Burns and Lynch shows that *any* mutual exclusion algorithm in this model must use at least n shared variables:

Theorem 5.1 (Burns, Lynch) *Any deadlock-free mutual exclusion algorithm (i.e., an algorithm that makes progress) requires n variables in the read/write atomic register model.*

The theorem becomes even more striking when compared with the impossibility result for the test-and-set model, as it gives a lower bound on the number of *variables*, not values. No restrictions apply to the algorithms, so the theorem applies to unfair or fair algorithms.

It is unclear where to begin proving such a theorem, since it must hold for all algorithms solving mutual exclusion. The intuition, difficult to make rigorous, is that if a system uses fewer than n shared variables, processes will not be able to record and send enough information to coordinate access to the critical resource.

We begin with a few definitions to make the notion of "communicating information" more explicit.

Definition Suppose process p performs a write to a variable v. The write is *obliterated* if another process overwrites v before v is read.

An obliterated write communicates no information to the other processes in a particular thread of execution. To make the thread of execution more explicit, we make a further definition:

Definition Let q be a state and h be a schedule (i.e., a list of the order in which processes take steps.) Process p_i is *hidden* from q by h if $h = h_1 h_2$ and p_i is in region R after h_1, and every write by p_i is obliterated during h_2.

If some processes are hidden in a certain schedule, other processes cannot be affected by the values of the obliterated writes. The following lemma formalizes this fact:

Lemma 5.2 *Let h be a finite schedule, and let P be a set of processes. For some state q_0, let $q = result(q_0, h)$ such that each $p_i \in P$ is hidden from q_0 by h. Then there is some state q' reachable from q_0 such that*

- *the values of the shared variables and the states of all processes not in P are the same in q' as in q, and*
- *if $p_i \in P$, then p_i is in R in q'.*

Proof: Starting in state q_0, run schedule h, omitting the steps of each $p_i \in P$ after it reaches its remainder region for the last time in h. (We know each eventually reaches R, since all processes in P are hidden.) We know that all writes by $p_i \in P$ are obliterated, either directly by some process $p_j \notin P$ or by some other process $p_k \in P$ whose writes are also either directly or indirectly obliterated by some process(es) not in P. Therefore, in the resulting state, the values of the shared variables will be the same as in q, and the states of the processes not in P will be the same as in q. See Figure 5.5 for a diagram. ■

During a computation, some variables may be obliterated in the next step.

Definition A variable v is *covered* by p_i in state q if p_i's next step is to write v. That is, p_i is ready to write v from state q.

Suppose a process writes a covered variable before it enters the critical region; this write could have disastrous consequences, since the the write could be immediately obliterated and, due to the loss of the "message", another process might be able to get to the critical region. The following lemma shows that, indeed, a process must write a non-covered variable before going to the critical region:

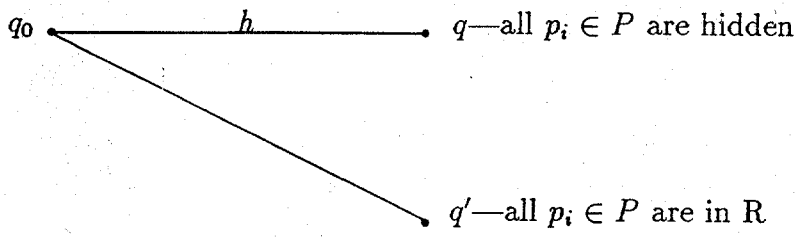
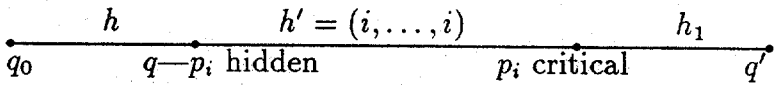


Figure 5.5: Hidden processes do not affect other processes. Here, if $p_i \in P$ are hidden during h, then q and q' agree in all respects except for the local state of the processes in P.

Lemma 5.3 *Let S be a system with more than 2 processes which solves deadlock-free mutual exclusion. Let h be a finite schedule, q_0 a state, and $q = result(q_0, h)$.*

Suppose p_i is hidden in the computation from q_0 by h. If p_i goes to its critical region on its own (by schedule $i, i, \ldots$) from state q, then in the computation from q, p_i must write some non-covered variable (i.e., not covered by any other process in state q.)

Proof: By contradiction. Suppose p_i gets to the critical region on its own from q without writing a non-covered variable; let this schedule be h'. Construct a new schedule $h'' = hh'h_1$,



where h_1 lets each other process take one step. Note that h_1 obliterates all writes by p_i, so p_i is hidden in the computation from q_0 by h''.

Now we use Lemma 5.2: There is some q'' reachable from q_0 which agrees with q' in the states of all other processes and all shared variables, but has p_i is in R. Since the system is deadlock-free, some process p_j can go to its critical region from q'' via some schedule s not involving p_i. But s applied to q' also lets p_j go to C, since from p_j's point of view, the state is the same. Thus, we can get processes p_i and p_j in C simultaneously, a violation of mutual exclusion. ■

Combining the definitions of "covered" and "hidden" will give us our goal. We call a variable *nullified* by a process if the process does not "communicate" with the other processes and covers the variable in its last step. Formally,

Definition Let q_0 be a state, h a schedule, and $q = result(q_0, h)$. A variable v is *nullified* by process p_i in the computation from q_0 by h if

1. p_i covers v in state q; and

2. p_i is hidden in the computation from q_0 by h.

Now we can show that, in any algorithm solving mutual exclusion, there is a finite schedule yielding n distinct nullified variables. Theorem 5.1 will then follow immediately.

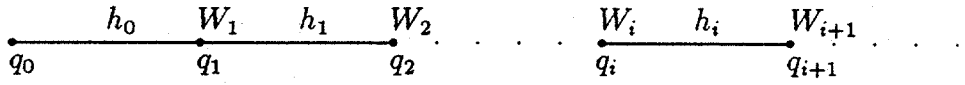
Lemma 5.4 *Let S be a system with more than 2 processes solving deadlock-free mutual exclusion. Let q_0 be any state in which all processes are in R.*

Then for every k, $1 \leq k \leq n$, there is a finite schedule h using only processes $p_1, \ldots, p_k$, such that k distinct variables are nullified by $p_1, \ldots, p_k$ in the computation from q_0 by h.

Proof: Proceed by induction on k. In the base case, $k = 1$. Since S is deadlock-free, running p_1 alone means that p_1 eventually reaches C; call this schedule h'. By Lemma 5.3, p_1 must write some non-covered variable during this computation. Stop h' just before p_1 writes *some* variable v (covered or non-covered) and call this schedule h. In this schedule from q_0, p_1 is hidden (since no variables are written) and p_1 covers v in $q_1 = result(q_0, h)$. Thus, one variable is nullified in the computation from q_0 by h.

In the induction case, assume the lemma holds for $k-1$. Then there is a finite schedule h_0 using only $p_1, \ldots, p_{k-1}$ such that $k-1$ distinct variables are nullified by $p_1, \ldots, p_{k-1}$. Let W_1 be the set of these variables, and $q_1 = result(q_0, h_0)$.

From this state q_1, one can create another schedule h_1' (using only processes $p_1, \ldots, p_{k-1}$) that first lets each of these processes write their covered variables, and then puts all of these processes into R (this is possible by the deadlock-free condition.) Applying the induction hypothesis again, we can get another h_1'' such that $k-1$ distinct variables are nullified by $p_1, \ldots, p_{k-1}$. Let W_2 be this set of variables, and $h_1 = h_1' h_1''$.



One can repeat this construction *ad infinitum*, as depicted above. Note that by the way we constructed h_i, each $p_1, \ldots, p_{k-1}$ is hidden from q_i by h_i.

Somehow, p_k must get involved in this computation so that it nullifies a distinct variable. To this end, we construct "side branches," schedules s_i which proceed from state q_i and involve only p_k. By Lemma 5.2, there is some state q' reachable from q_{i-1} such that

- $p_1, \ldots, p_{k-1}$ are all in R, and
- the values of all shared variables, and the state of p_k, is the same in q' as in q_i.

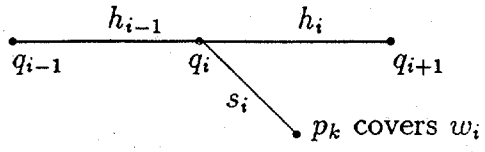


Figure 5.6: The construction of side branches. Note that s_i only involves process p_k.

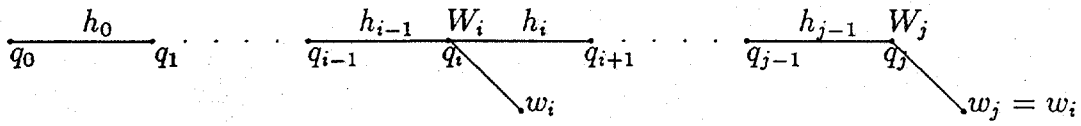


Figure 5.7: The construction of the schedule h. Process p_k covers $w_i = w_j$ in the two side branches depicted.

In other words, q' "looks like" q_i from the perspective of p_i. From q', p_k must be able to reach C on its own, since the algorithm is deadlock-free and all processes are in R. Thus, p_k can also reach C on its own from q_i, since q_i "looks like" q'. Call such a schedule s_i'. Note that p_k is trivially hidden from q_{i-1} by h_{i-1}, since we do not let it take a step. By Lemma 5.3, p_k must write some variable v during the computation from q_i via s_i', where v is not covered by any process in $p_1, \ldots, p_{k-1}$. Thus, $v \notin W_i$. Let s_i be the shortest schedule from q_i where p_k covers some variable $w_i \notin W_i$ (see Figure 5.6.)

We're almost done, since after running s_i each of $p_1, \ldots, p_k$ covers a distinct variable. We still must show that p_k is hidden. We use a combinatorial trick. Choose two different states q_i and q_j with $w_i = w_j$; we know that these states exist by a pigeonhole argument, since we can construct a very long chain (length $> k$) of the q_i's (see Figure 5.7). Create a schedule

$$h = h_0 \ldots h_{i-1} s_i h_i \ldots h_{j-1}.$$

Note that p_k could only write to the variables in W_i during s_i. Thus, p_k becomes hidden during h_i, since all variables in W_i are covered, and p_k stays hidden from h_i on, since it does not take any steps. Now $w_i \notin W_j$, since $w_i = w_j \notin W_j$. Since $p_1, \ldots, p_{k-1}$ nullify $k-1$ distinct variables in W_j during h_{j-1}, and since p_k nullifies $w_i \notin W_j$, the k processes nullify k different variables. ■

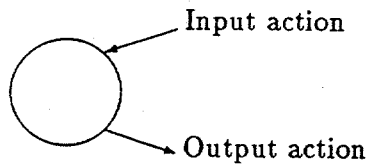


Figure 5.8: An I/O Automaton.

5.4 Introduction to I/O Automata

In the discussion of mutual exclusion, we have seen many different models, each with its own advantages and disadvantages. A single model of distributed computation, one that could capture the aspects of these models, would certainly make things less complicated: descriptions of algorithms would become uniform, and proofs of correctness would be easier to certify.

One such model is the *I/O automaton*, a model designed for asynchronous distributed settings. Informally, I/O automata are a special form of automata (not necessarily finite state) with three kinds of actions: input, output, and internal (see Figure 5.8). Problems in this model are phrased as sets of sequences of *external* actions, input or output actions. An automaton *solves* a problem if the automaton's set of sequences, or *behaviors*, are a subset of the problem's.

Somewhat more formally, let S be a set of *actions*; an *action signature* is a triple $(in(S), out(S), int(S))$ with $in(S)$, $out(S)$, and $int(S)$ disjoint sets drawn from S. These three sets represent the input, output, and internal actions respectively. *External actions* are input or output actions, or in other words, $ext(S) = in(S) \cup out(S)$.

Definition An *I/O automaton* A is a quintuple composed of

- $sig(A)$, an action signature;
- $states(A)$, a set of states;
- $start(A)$, a set of distinguished start states;
- $steps(A) = \{(s', \pi, s) \mid s', s \in states(A), \pi \text{ is an action } \}$; and
- $part(A)$, an equivalence relation on $local(S) = int(S) \cup out(S)$.

The I/O Automaton model is discussed in detail in Lecture 6. Automata may be nondeterministic, and are always *input-enabled*, *i.e.*, they must be able to accept any input at any time. This is different from other models, such as CSP, that allow inputs to be blocked by a component. Another important definition is that of an *execution*, an alternating sequence states and actions such that each state-action-state triple is a step of the automaton.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 6: September 29

Lecturer: Nancy Lynch | *Scribe: John Keen*

6.1 I/O Automata

This lecture is based on 'I/O Automata: A Model for Discrete Event Systems' by N. Lynch, MIT/LCS/TM-351. The paper appears here in edited form.

The *input/output automaton* model has recently been defined, in [LynchT87], as a tool for modelling concurrent and distributed discrete event systems of the sorts arising in computer science. Since its introduction, the model has been used for describing and reasoning about several different types of systems, including network resource allocation algorithms, communication algorithms, concurrent database systems, shared atomic objects, and dataflow architectures. The simplicity and generality of the model and its similarities with other new models ([RamadgeW85], [ChandyM88]). suggest that it will prove useful in other application areas, such as control theory and manufacturing.

6.1.1 Overview of the Model

I/O automata provide an appropriate model for discrete event systems consisting of concurrently-operating components. The components, as well as the entire system, may be 'reactive' in the sense that they interact with their environments in an ongoing manner (rather than, say, simply accepting an input, computing a function of that input and halting). Although I/O automata can be used to model synchronous systems, they are best suited for modelling systems in which the components operate asynchronously.

Each system component is modelled as an 'I/O automaton', which is a mathematical object somewhat like a traditional automaton. However, an I/O automaton need not be finite-state, but can have an infinite state set. The actions of an I/O automaton are classified as either 'input', 'output', or 'internal'. The automaton generates output and internal actions autonomously, and transmits output actions instantaneously to its environment. In contrast, the automaton's input is generated by the environment and transmitted instantaneously to the automaton. Our distinction between input and other actions is fundamental, based on who determines when the action is performed: an automaton can establish restrictions on when it will perform an output or internal action, but it is unable to block the performance of an input action.

The fact that our automata are unable to block inputs distinguishes our model from Hoare's CSP (Communicating Sequential Processes) [Hoare84]. There, input blocking is used

for two purposes: as a way of eliminating undesirable inputs, and as a way of blocking the activity of the environment. Our model does not have any way of blocking the environment, but does have other ways of coping with bad inputs. For example, suppose that we wish to constrain the behavior of an automaton only in case the environment observes certain restrictions on the production of inputs. Instead of requiring the automaton to block the bad inputs, we permit these inputs to occur; however, we may permit the automaton to exhibit arbitrary behavior in case they do. Alternatively, we may require the automaton to detect bad inputs and respond to them with error messages. Thus, we have simple ways of describing input restrictions, without including input-blocking in the model.

I/O automata may be nondeterministic, and indeed the nondeterminism is an important part of the model's descriptive power. Describing algorithms as nondeterministically as possible tends to make results about the algorithms quite general, since many results about nondeterministic algorithms apply a fortiori to all algorithms obtained by restricting the nondeterministic choices. Moreover, the use of nondeterminism helps to avoid cluttering algorithm descriptions and proofs with inessential details.

I/O automata can be composed to yield other I/O automata. Our composition operator connects each output action of one automaton with input actions of any number (usually one) of other automata. In the resulting system, an output action is thus generated autonomously by one component and instantaneously transmitted to all the other components having the same action as an input. All such components are passive recipients of the input, and take steps simultaneously with the output step. As in CSP, we use simultaneous performance of actions to synchronize components, but we permit only one component to determine when the action occurs.

Since I/O automata are intended to model complex systems with any number of primitive components, each automaton comes equipped with an abstract notion of 'component'; formally, these components are described by an equivalence relation on the automaton's output and internal actions, where all the actions in one equivalence class are to be thought of as under the control of the same primitive system component.

When I/O automata are run, they generate 'executions' (alternating sequences of states and actions). Among all the executions of an automaton, we are primarily interested in the 'fair' executions - those that permit each of the automaton's primitive components to have infinitely many chances to perform output or internal actions. The fair executions of an automaton give rise to the 'fair behaviors' of the automaton - the subsequences of the fair executions that consist of external (i.e., input and output) actions. It is this set of sequences that we believe embodies the interesting behavior of an I/O automaton; thus, our semantics is a 'trace' semantics. The set of fair behaviors of an I/O automaton can consist of both finite and infinite sequences of actions, and is not necessarily closed under the operation of taking prefixes.

A 'problem' to be solved by an I/O automaton is formalized essentially as an arbitrary set of (finite and infinite) sequences of external actions. Our notion of what it means for an

automaton to 'solve' a problem is particularly simple: essentially, an automaton is said to 'solve' a problem P provided that its set of fair behaviors is a subset of P. It might not be obvious to the reader that this definition is nontrivial; for example, if an automaton had no fair behaviors, then our definition would say that it is a solution to every problem. However, this anomaly does not arise, since our automaton definitions imply that every automaton has a nonempty set of fair behaviors (note that even a trivial automaton having no actions at all has one fair behavior - the empty sequence of actions). The fact that inputs are always allowed gives another reason why our definition of solving a problem is nontrivial: for every possible pattern of inputs that might arrive from the environment, the automaton is required to provide some response such that the resulting sequence of actions is in the problem set P. That is, the automaton is required to respond appropriately to every possible input pattern.

The model permits description of algorithms and systems at different levels of abstraction. Abstraction mappings are defined, mapping automata that include implementation detail to more abstract automata that suppress some of the detail. Such mappings can be used as aids in correctness proofs for algorithms: if automaton A is an image of B under an appropriate abstraction mapping and A solves problem P, then B also solves P.

The model allows very careful and readable descriptions of particular concurrent algorithms. We have developed a simple language for describing automata, based on 'Precondition' and 'Effect' specifications for actions. This notation, similar to Dijkstra's 'guarded commands' has proved sufficient for describing all algorithms we have attempted so far. However, the model does not constrain the user to describe all automata in this manner; for example, the model is general enough to serve also as a formal basis for languages that include more elaborate constructs for sequential flow of control.

Our model also allows precise statement of the problems that are to be solved by modules in concurrent systems. As described above, such problems are formulated as sets of finite and infinite sequences of external actions. We have not so far developed any particular language or notation for describing such sets, but have used a variety of notations (e.g. temporal logic or generating automata) as they have seemed convenient. Our model is general enough to serve as a semantic model for many different languages for describing sets of action sequences.

The model can be used as a formal basis for algorithm correctness proofs - proofs that particular algorithms solve particular problems in the sense described above. In fact, a current major thrust of our research involves producing correctness proofs for substantial-sized and complex concurrent algorithms. We use a variety of techniques for such proofs, primarily based on notions of composition and abstraction. In every case, we try to utilize the modularity that is suggested by informal descriptions of the algorithm in our formal correctness proofs. So far, our proofs have been done by hand, but it appears that machine-checking of some of our proofs might be possible using current automatic proof technology.

The model can also be used for carrying out complexity analysis, proving upper and lower bounds on the complexity of solving particular problems, and proving impossibility results.

6.1.2 Definitions and Basic Results

This section contains some of the basic definitions and results about the model. This material is adapted from [LynchT87].

Actions and Action Signatures

We assume a universal set of *actions*. Sequences of actions are used in this work, for describing the behavior of modules in concurrent systems. Since the same action may occur several times in a sequence, it is convenient to distinguish the different occurrences. Thus, we refer to a particular occurrence of an action in a sequence as an *event*.

The actions of each automaton are classified as either 'input', 'output', or 'internal'. The distinctions are that input actions are not under the automaton's control, output actions are under the automaton's control and externally observable, and internal actions are under the automaton's control but not externally observable. In order to describe this classification, each automaton comes equipped with an 'action signature'.

An *action signature* S is an ordered triple consisting of three pairwise-disjoint sets of actions. We write *in(S)*, *out(S)* and *int(S)* for the three components of S, and refer to the actions in the three sets as the *input actions*, *output actions* and *internal actions* of S, respectively. We let *ext(S)* = in(S) $\cup$ out(S) and refer to the actions in ext(S) as the *external actions* of S. Also, we let *local(S)* = out(S) $\cup$ int(S), and refer to the actions in local(S) as the *locally-controlled actions* of S. Finally, we let acts(S) = in(S) $\cup$ out(S) $\cup$ int(S), and refer to the actions in acts(S) as the *actions* of S. An *external action signature* is an action signature consisting entirely of external actions, that is, having no internal actions. If S is an action signature, then the *external action signature* of S is the action signature *extsig(S)* = (in(S),out(S),ϕ), i.e., the action signature that is obtained from S by removing the internal actions.

Input/Output Automata

Now we are ready to define the basic component of our model. An *input/output automaton* A (also called an *I/O automaton* or simply an *automaton*) consists of five components:

- an action signature *sig(A)*,
- a set *states(A)* of *states*,
- a nonempty set *start(A)* $\subseteq$ states(A) of *start states*,
- a transition relation *steps(A)* $\subseteq$ states(A) $\times$ acts(sig(A)) $\times$ states(A), with the property that for every state s' and input action π there is a transition (s',π,s) in steps(A), and

- an equivalence relation *part(A)* on local(sig(A)), having at most countably many equivalence classes.

We refer to an element (s',π,s) of steps(A) as a *step* of A. The step (s',π,s) is called an *input step* of A if π is an input action. *Output steps*, *internal steps*, *external steps* and *locally-controlled steps* are defined analogously. If (s',π,s) is a step of A, then π is said to be *enabled* in s'. Since every input action is enabled in every state, automata are said to be *input-enabled*. The input-enabling property means that the automaton is not able to block input actions. The partition part(A) is what was described in the introduction as an abstract description of the 'components' of the automaton. It is used to define fairness.

An *execution fragment* of A is a finite sequence $s_0,\pi_1,s_1,\pi_2,\ldots,\pi_n,s_n$ or an infinite sequence $s_0,\pi_1,s_1,\pi_2,\ldots,\pi_n,s_n,\ldots$ of alternating states and actions of A such that (s_i,π_{i+1},s_{i+1}) is a step of A for every i. An execution fragment beginning with a start state is called an *execution*. We denote the set of executions of A by *execs(A)*, and the set of finite executions of A by *finexecs(A)*. A state is said to be *reachable* in A if it is the final state of a finite execution of A.

A *fair execution* of an automaton A is defined to be an execution α of A such that the following conditions hold for each class C of part(A).

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many events from C, or else α contains infinitely many occurrences of states in which no action of C is enabled.

Thus, a fair execution gives 'fair turns' to each class of part(A). We denote the set of fair executions of A by *fairexecs(A)*.

The *schedule* of an execution fragment α of A is the subsequence of α consisting of actions, and is denoted by *sched(α)*. We say that β is a *schedule* of A if β is the schedule of an execution of A. We denote the set of schedules of A by *scheds(A)* and the set of finite schedules of A by *finscheds(A)*. We say that β is a *fair schedule* of A if β is the schedule of a fair execution of A and we denote the set of fair schedules of A by *fairscheds(A)*. The *behavior* of an execution or schedule α of A is the subsequence of α consisting of external actions, and is denoted by *beh(α)*. We say that β is a *behavior* of A if β is the behavior of an execution of A. We denote the set of behaviors of A by *behs(A)* and the set of finite behaviors of A by *finbehs(A)*. We say that β is a *fair behavior* of A if β is the behavior of a fair execution of A and we denote the set of fair behaviors of A by *fairbehs(A)*.

Schedule Modules

In order to describe problems to be solved by automata, we need to describe sets of sequences. More precisely, a problem will be specified by a pair consisting of an action signature and

a set of sequences over the actions in that signature. (In most interesting cases, the action signature will be an external action signature.) The mathematical object used to describe a problem is called a 'schedule module'.

A *schedule module* H consists of two components:

- an action signature *sig(H)*, and
- a set *scheds(H)* of *schedules*.

Each schedule in scheds(H) is a finite or infinite sequence of actions of H. Let *finscheds(H)* denote the set of finite members of scheds(H).

The *behavior* of a schedule β of H is the subsequence of β consisting of external actions, and is denoted by *beh(β)*. We say that β is a *behavior* of H if β is the behavior of a schedule of H. We denote the set of behaviors of H by *behs(H)* and the set of finite behaviors of H by *finbehs(H)*. We extend the definitions of fair schedules and fair behaviors to schedule modules in a trivial way, letting *fairscheds(H)* = scheds(H) and *fairbehs(H)* = behs(H).

We use the term *module* to designate either an automaton or schedule module. If M is a module, we sometimes write *acts(M)* as shorthand for acts(sig(M)), and likewise for in(M), out(M), etc. If β is any sequence of actions and M is a module, we write $\beta \mid$ M for $\beta \mid$ acts(M).

There are several natural schedule modules that we often wish to associate with an automaton. They correspond to the automaton's schedules, finite schedules, fair schedules, behaviors, finite behaviors and fair behaviors. For each automaton A, let *Scheds(A)*, *Finscheds(A)*) and *Fairscheds(A)* be the schedule modules having action signature sig(A) and having schedules scheds(A), finscheds(A) and fairscheds(A), respectively. Also, for each module M, let *Behs(M)*, *Finbehs(M)* and *Fairbehs(M)* be the schedule modules having action signature extsig(M) and having schedules behs(M), finbehs(M) and fairbehs(M), respectively. (Here and elsewhere, we follow the convention of denoting sets of schedules with lower case names and corresponding schedule modules with corresponding upper case names.)

Solving Problems

Now we are ready to define our notion of 'solving'. This notion is intended for describing the way in which particular automata solve particular problems (formalized as schedule modules). However, it is convenient to state the definition more generally. Let M and M' be modules (i.e., either automata or schedule modules) with the same external action signature. Then M' is said to *solve* M if fairbehs(M') $\subseteq$ fairbehs(M). Note that M does not need to exhibit *all* of the behaviours in fairbehs(M); merely a subset is sufficient. You may think of M as defining a set of constraints, which the behaviour of M' must obey.

In the most interesting case, M' is an automaton and M is a schedule module. However, the more general formulation allows us to carry out proofs in several stages: in order to show that an automaton solves a problem, we can show that the automaton 'solves' another

automaton, which in turn solves another automaton, and so on, until some final automaton solves the problem. A variety of techniques can be used to show that an automaton M' solves a schedule module M; we will mention some of these below.

Implementation

One way of showing that one module solves another is to use an intermediate result about inclusion for the sets of finite behaviors. Thus, we define an analog of the 'solving' definition for finite behaviors only. Let M and M' be modules with the same external action signature. Then M' is said to *implement* M if finbehs(M') $\subseteq$ finbehs(M).

It is often possible to show that one automaton implements another using a mapping between automaton states. Suppose A and B are automata with the same external action signature, and suppose f is a mapping from states(A) to the power set of states(B). The mapping f is said to be a *possibilities mapping* from A to B if the following conditions hold:

1. For every start state s of A, there is a start state t of B such that t $\in$ f(s).
2. For every reachable state s' of A, every step (s',π,s) of A, and every reachable state $t' \in f(s')$ of B:
   (a) If $\pi \in$ acts(B), then there is a step (t',π,t) of B such that t $\in$ f(s).
   (b) If $\pi \notin$ acts(B), then t' $\in$ f(s).

Lemma 6.1 *Suppose that A and B are automata with the same external action signature and there is a possibilities mapping from A to B. Then A implements B.*

Figure 6.1 illustrates the idea of one I/O automaton mapping to another one. It shows the two cases: $\pi \in$acts(B) and $\pi \notin$acts(B).

It is possible to show that one module M' solves another module M using this lemma together with additional results showing correspondences between fairness properties of M and M'. Some such additional results are given in [LynchT87] and [WelchLL88].

Composition

The most useful way of combining I/O automata is by means of a composition operator, as defined in this subsection. The intuition behind the composition of several I/O automata is as follows. We do not allow a particular action to be an output of more than one IOA, and we do not allow an internal action of any IOA to be an input, output or internal action of any other IOA (ie: internal actions must be unique). Several IOA's may have a common input action, but for any particular action, the number of IOA's having it as an input must be finite. If an output action is an input to one or more other IOA's, we conceptually "hook up" the output to these other inputs.

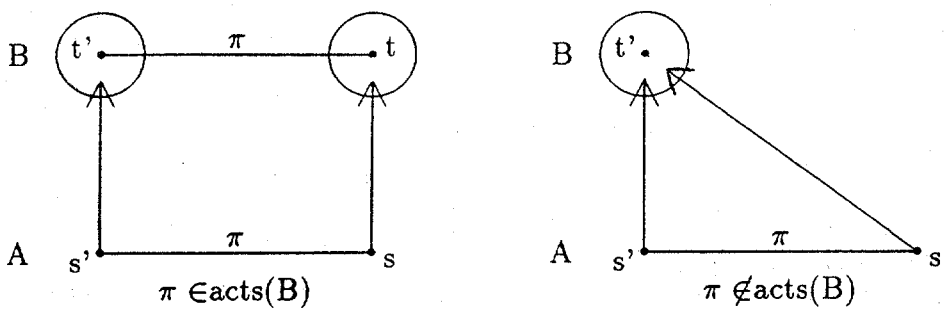


Figure 6.1: Mapping of I/O Automaton A to IOA B. Circles indicate equivalence classes of states of B.

Composition of Action Signatures

Let I be an index set that is at most countable. A collection $\{S_i\}_{i \in I}$ of action signatures is said to be *strongly compatible* if for all i, j $\in$ I, i $\neq$ I, we have

1. $\text{out}(S_i) \cap \text{out}(S_j) = \phi$,
2. $\text{int}(S_i) \cap \text{acts}(S_j) = \phi$, and
3. no action is in $\text{acts}(S_i)$ for infinitely many i.

Thus, no action is an output of more than one signature in the collection, and internal actions of any signature do not appear in any other signature in the collection.

A collection of action signatures is said to be *compatible* if it satisfies the first two of the three listed properties. Some of the results below follow simply from compatibility, while others require strong compatibility. Here, we simplify matters by considering the stronger definition only. The consequences of the two definitions are described more carefully in [LynchT87].

The *composition* $S = \prod_{i \in I} S_i$ of a collection of strongly compatible action signatures $\{S_i\}_{i \in I}$ is defined to be the action signature with

- $\text{in}(S) = \cup_{i \in I}\text{in}(S_i)$ - $\cup_{i \in I}\text{out}(S_i)$,
- $\text{out}(S) = \cup_{i \in I}\text{out}(S_i)$, and
- $\text{int}(S) = \cup_{i \in I}\text{int}(S_i)$.

Thus, output actions are those that are outputs of any of the component signatures, and similarly for internal actions. Input actions are any actions that are inputs to any of the component signatures, but outputs of no component signature.

Composition of Automata

A collection $\{M_i\}_{i \in I}$ of modules is said to be *strongly compatible* if their action signatures are strongly compatible. The *composition* $A = \prod_{i \in I} A_i$ of a strongly compatible collection of automata $\{A_i\}_{i \in I}$ has the following components:

- sig(A) = $\prod_{i \in I} sig(A_i)$,
- states(A) = $\prod_{i \in I} states(A_i)$,
- start(A) = $\prod_{i \in I} start(A_i)$,
- steps(A) is the set of triples $(\vec{s_1}, \pi, \vec{s_2})$ such that for all i∈I, if $\pi \in$ acts(A_i) then $(\vec{s_1}[i], \pi, \vec{s_2}[i]) \in$ steps(A_i), and if $\pi \notin$ acts(A_i) then $\vec{s_1}[i] = \vec{s_2}[i]$,
- part(A) = $\cup_{i \in I}$ part(A_i).

The second and third components listed are just ordinary Cartesian products, while the first component uses a previous definition.

Note that we use the notation $\vec{s}[i]$ to denote the i^{th} component of the state vector $\vec{s}$.

Since the automata A_i are input-enabled, so is their composition, and hence their composition is an automaton. Each step of the composition automaton consists of all the automata that have a particular action in their signatures performing that action concurrently, while the automata that do not have that action in their signatures do nothing. In composing automata, one would like to still be fair to the separate loci of control, therefore the partition for the composition is formed by taking the union of the partitions for the components. Thus, a fair execution of the composition gives fair turns to all of the classes within all of the component automata. In other words, all component automata in a composition continue to act autonomously. If $\alpha = \vec{s_0}\pi_1\vec{s_1}$ ... is an execution of A, let $\alpha|A_i$ be the sequence obtained by deleting $\pi_j\vec{s_j}$ when π_j is not an action of A_i, and replacing the remaining $\vec{s_j}$ by $\vec{s_j}[i]$.

The following basic results relate executions, schedules and behaviors of a composition to those of the automata being composed. The first result says that the projections of executions of a composition onto the components are executions of the components, and similarly for schedules, etc. The parts of this result dealing with fairness depend on the fact that at most one component automaton can impose preconditions on each action.

Lemma 6.2 *Let $\{A_i\}_{i \in I}$ be a strongly compatible collection of automata, and let $A = \prod_{i \in I} A_i$. If $\alpha \in$ execs(A) then $\alpha|A_i \in$ execs(A_i) for all i∈I. Moveover, the same result holds for finexecs, fairexecs, scheds, finscheds, fairscheds, behs, finbehs and fairbehs in place of execs.*

Certain converses of the preceding lemma are also true. The following lemma says that executions of component automata can be patched together to form an execution of the composition.

Lemma 6.3 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \prod_{i\in I} A_i$. For all $i\in I$, let α_i be an execution of A_i. Suppose β is a sequence of actions in ext(A) such that $\beta|A_i = beh(\alpha_i)$ for every i. Then there is an execution α of A such that $\beta = beh(\alpha)$ and $\alpha_i = \alpha|A_i$ for all i. Moreover, if α_i is a fair execution of A_i for all i, then α may be taken to be a fair execution of A.*

Similarly, schedules or behaviors of component automata can be patched together to form schedules or behaviors of the composition.

Lemma 6.4 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata, and let $A = \prod_{i\in I} A_i$. Let β be a sequence of actions in acts(A). If $\beta|A_i \in scheds(A_i)$ for all $i \in I$, then $\beta \in scheds(A)$. Moreover, the same result holds for fairscheds, behs and fairbehs in place of scheds.*

The previous lemmas are often useful in proving that certain automata solve certain problems. In particular, sometimes correctness conditions are formulated to say that every behavior of an automaton is also a behavior of a given composition A. One way of showing that a given sequence of actions is a behavior of A is by first showing that its projections are behaviors of the components of A and then appealing to the preceding lemmas.

Composition of Schedule Modules

Corresponding to our composition operator for automata, we also define a composition operator for schedule modules. The *composition* $\mathrm{H} = \prod_{i\in I} H_i$ of strongly compatible schedule modules $\{H_i\}_{i\in I}$ is defined to be the schedule module with

- $\mathrm{sig(H)} = \prod_{i\in I} \mathrm{sig}(H_i)$,
- scheds(H) is the set of sequences β of actions of H such that $\beta \mid H_i$ is a schedule of H_i for every $\mathrm{i} \in \mathrm{I}$.

The following lemma shows how composition of schedule modules corresponds to composition of automata.

Lemma 6.5 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Then $Scheds(A) = \prod_{i\in I} Scheds(A_i)$, $Fairscheds(A) = \prod_{i\in I} Fairscheds(A_i)$, $Behs(A) = \prod_{i\in I} Behs(A_i)$ and $Fairbehs(A) = \prod_{i\in I} Fairbehs(A_i)$.*

Preserving Properties

Although automata in our model are unable to block input actions, it is often convenient to restrict attention to behaviors in which the environment obeys certain 'well-formedness' restrictions. A useful way of discussing such restrictions is in terms of the notion that a module 'preserves' a property of behaviors: as long as the environment does not violate the property, neither does the module. Such a notion is primarily interesting for properties that are 'prefix-closed'.

A set of sequences P is *prefix-closed* provided that whenever $\alpha \in$ P and β is a prefix of α, it is also the case that $\beta \in$ P. A module M is said to be *prefix-closed* provided that behs(M) is prefix-closed. Let M be any prefix-closed module and let P be a prefix-closed set of sequences of actions in ext(M). We say that M *preserves* P if $\beta = \beta'\pi \in$ finbehs(M), $\pi \in$ out(M) and $\beta' \in$ P together imply that $\beta \in$ P. Thus, if a module preserves a property P, the module is not the first to violate P: as long as the environment only provides inputs such that the cumulative behavior satisfies P, the module will only perform outputs such that the cumulative behavior satisfies P.

Hiding Actions

Here we define an operator that 'hides' some of the output actions of a module by converting them to internal actions. We begin with a hiding operator on action signatures: if S is an action signature and Σ is a subset of out(S), define $hide_\Sigma$(S) = S', where in(S') = in(S), out(S') = out(S) - Σ and int(S') = int(S)$\cup\Sigma$. Now we use the hiding operator on signatures to define a hiding operator for automata and schedule modules: if M is a module with signature S, and $\Sigma \subseteq$ out(S), then let $hide_\Sigma$(M) be the module M' that coincides with M except that sig(M') = $hide_\Sigma$(sig(M)).

6.1.3 Candy Machines

In this section, we illustrate many of the preceding definitions using examples of simple candy machines. (This class of examples is popular in the CSP literature, so this choice should facilitate comparison of the models.) These examples show how our model is used to define simple nondeterministic processes. They also show how problems can be stated, and how it can be proved that certain automata solve certain problems. Finally, they show how processes can interact in the model, although the style of interaction is very simple (normally a strict alternation of button pushes and candy dispensations).

Candy Machines

In this subsection, we describe three specific candy machines as I/O automata. Candy machine model CM-1 has the following action signature.

Input actions:	PUSH1, PUSH2
Output actions:	SKYBAR, HEATHBAR, ALMONDJOY
Internal actions:	none

We will sometimes abbreviate the two push actions as 1 and 2, respectively, and the three dispensation actions as S, H and A. The state of CM-1 consists of one variable 'button_pushed', which takes on values 0, 1 and 2, initially 0. Next we describe the transition relation of CM-1. It should not be hard for the reader to translate the given description into a transition relation: (s',π,s) is a step of the automaton exactly if the precondition of π (if any) is satisfied in s' and s is a possible result of running the code in π's 'Effect' starting from s'.

PUSH1
Effect: button_pushed := 1

PUSH2
Effect: button_pushed := 2

SKYBAR
Precondition: button_pushed = 1
Effect: button_pushed := 0

HEATHBAR
Precondition: button_pushed = 2
Effect: button_pushed := 0

ALMONDJOY
Precondition: button_pushed = 2
Effect: button_pushed := 0

Thus, when the customer pushes button 1, CM-1 can dispense a SKYBAR. When the customer pushes button 2, CM-1 can dispense either a HEATHBAR or an ALMONDJOY, but not both. The partition for this automaton, part(CM-1), is defined to group together ALMONDJOY and HEATHBAR and to keep SKYBAR in a singleton set.

Candy machine model CM-2 is identical to CM-1 except that its HEATHBAR action has Precondition 'false'. This candy machine never dispenses HEATHBARs, but is able to dispense SKYBARs and ALMONDJOYs. Model CM-3 is identical to CM-1 except that all three candy dispensation actions have Precondition 'false'. That is, it never dispenses candy. As one might expect, it is not a very useful candy machine from the point of view of the customer.

Specifications for Candy Machine Behavior

Now we describe some interesting notions of correct candy machine behavior.

Safe Candy Machine Behavior

Some basic requirements for a candy machine can be described by the schedule module SAFE-CM. SAFE-CM has the same action signature as CM-1, and has as its set of schedules the set of sequences over the symbols 1,2,S,H,A satisfying the following condition: every S is immediately preceded by a 1, and every A or H is immediately preceded by a 2.

In order to show that CM-1 is a safe candy machine, i.e., that it solves the problem described by SAFE-CM, we must show that all fair behaviors of CM-1 satisfy the given requirement. Note that this requirement, (as usual for safety requirements) holds for an infinite sequence if and only if it holds for all finite prefixes of the infinite sequence. Therefore, it suffices to show that all finite behaviors of CM-1 satisfy the given requirement.

We proceed by induction on the length of a behavior, using an inductive hypothesis that characterizes the state of CM-1 in terms of the preceding events, i.e., button_pushed = 1 if the last event in the sequence is PUSH1, 2 if the last event in the sequence is PUSH2, and 0 otherwise (i.e., if the sequence is empty, or if the last event is a dispensation event). The inductive step considers cases based on the five possible events. For instance, if SKYBAR occurs, its Precondition implies that button_pushed = 1 just prior to the dispensation; thus, the immediately preceding symbol in the sequence is 1, as needed. The other cases are similar. It follows that CM-1 is a safe candy machine.

It is also easy to see that CM-2 is a safe candy machine. However, saying that CM-1 and CM-2 are safe candy machines is not really saying enough, since the same is also true for CM-3. CM-3's fair behaviors are just the finite and infinite sequences of 1's and 2's, which trivially satisfy the required condition. Although CM-3 is a safe candy machine, it is not a very interesting one. Therefore, we will give a stronger specification below.

Well-Formedness

In discussing correct candy machine behavior, it is helpful to consider certain 'well-formedness' conditions on the interaction between the machine and its environment. For example, we may want to restrict attention to interactions in which push and dispensation events alternate strictly. Define a sequence of candy machine actions to be *well-formed* if it consists of alternating input and output (push and dispensation) actions, starting with an input action. Notice that CM-1 has behaviors, in fact fair behaviors, that are not well-formed, e.g. 11S11S... is a non-well-formed fair behavior of CM-1. This is not surprising, since CM-1 does not (in our model) have the power to insure that its environment preserves well-formedness. However, it is easy to see that any safe candy machine, including CM-1, preserves well-formedness, according to the definition of 'preserves' given in Section 6.3.

Live Candy Machine Behavior

A stronger set of requirements than SAFE-CM can be described by the schedule module LIVE-CM. LIVE-CM has the same action signature as CM-1. Its set of sequences are those that are safe candy machine sequences and that in addition satisfy the following condition: 'If the sequence is well-formed, then every push event has a later dispensation event.'

CM-3 is not a live candy machine, because it has fair behaviors, such as the sequence with the single element 1, that do not satisfy this condition. (This sequence satisfies the well-formedness hypothesis, but does not satisfy the liveness conclusion.) On the other hand, CM-1 is a live candy machine, which we can prove as follows. Suppose not; then there is a fair behavior of CM-1 that is well-formed and that contains a push event that is not followed by any later dispensation event. By well-formedness, the only possibility is that the sequence is finite and ends with the given push event. Say, for example, that the push event is PUSH1. Then by the state characterization given above, the state after the given schedule has button_pushed = 1. Then the SKYBAR dispensation action is enabled in this state. But the definition of a fair execution implies that no action of CM-1 can be enabled in the final state, which yields a contradiction.

CM-2 is also a live candy machine, even though it has less nondeterminism than CM-1. The proof is similar to that for CM-1.

For the reasons discussed in Section 6.2, LIVE-CM does not admit trivial solutions. Anything that satisfies the specification must be able to respond to any pattern of pushes (since it is an I/O automaton, with the input-enabling condition). Moreover, responses have to be safe, and if the pushes arrive in a well-formed way, responses must in fact be made.

One might ask the technical question whether it might be reasonable to eliminate the well-formedness hypothesis in the live candy machine behavior specification. If we did this, then we might arrive at a stronger specification for a live candy machine, one that requires that the machine must always issue candy sometime after each push, regardless of whether the pushes happen in a well-formed manner. While this might be a reasonable requirement for a candy machine, CM-1 does not satisfy it. For consider the (non-well-formed) behavior 121212... of CM-1. This contains push events that are not followed by dispensation events. However, we claim it is a fair behavior of CM-1, since each class in the partition part(CM-1), {S} and {A,H}, has infinitely many points in the sequence at which no action in that class is enabled. (It might be helpful for the reader to imagine that there are two 'processes' inside the candy machine, where process 1 is in charge of dispensing SKYBARS and process 2 is in charge of dispensing ALMONDJOYS and HEATHBARS. Every time process 1 tries to perform its task, it happens that the value of button_pushed is 2, so it cannot do anything. Similarly, every time process 2 tries to perform its task, the value of button_pushed is 1. So neither process can cause any output to occur.) Since we have exhibited a fair behavior of CM-1 that contains a push but no later dispensation, CM-1 does not satisfy the proposed stronger specification.

Customers

We now describe particular customers that might interact with a candy machine. It is convenient also to describe such customers as I/O automata also. Customer CUST-1 continues to request candy bars ad infinitum, nondeterministically choosing which button to push. CUST-1's action signature is the 'complement' of that of the candy machines':

Input actions:	SKYBAR, HEATHBAR, ALMONDJOY
Output actions:	PUSH1, PUSH2
Internal actions:	none

The state of CUST-1 consists of one variable 'waiting', which takes on values 'yes' and 'no', initially 'no'. CUST-1's actions are as follows.

SKYBAR
Effect: waiting := no

HEATHBAR
Effect: waiting := no

ALMONDJOY
Effect: waiting := no

PUSH1
Precondition: waiting = no
Effect: waiting := yes

PUSH2
Precondition: waiting = no
Effect: waiting := yes

The partition part(CUST-1) puts PUSH1 and PUSH2 together in one equivalence class. It is easy to see that CUST-1 preserves well-formedness; in fact, it never pushes unless all previous pushes have been followed by dispensations. Also, in any well-formed fair behavior, after any dispensation event, CUST-1 eventually pushes a button once again.

Customer CUST-2 is somewhat more selective than CUST-1. It pushes button 2 repeatedly just until the machine dispenses a HEATHBAR. Then it pushes button 1 forever. Formally, CUST-2 has another variable 'heathbar_received' in its state in addition to 'waiting'. This variable takes on values 'yes' and 'no', initially 'no'. The actions of CUST-2 that differ from those of CUST-1 are as follows.

HEATHBAR
Effect: waiting := no; heathbar_received := yes

PUSH1
Precondition: waiting = no; heathbar_received = yes
Effect: waiting := yes

PUSH2
Precondition: waiting = no; heathbar_received = no
Effect: waiting := yes

It is easy to show that CUST-2 implements CUST-1, using a possibilities mapping f that maps each state s of CUST-2 to the singleton set containing the state of CUST-1 that only contains the 'waiting' variable of s. In fact, it can be shown that CUST-2 solves CUST-1, according to our formal definition of 'solves'. A straightforward proof can be based directly on the definition of fair execution and the fact that for every state s of CUST-2, some output action is enabled in s for CUST-2 exactly if some output action is enabled in f(s) for CUST-1.

Customer CUST-3 is similar to CUST-1 except that it is required eventually to take a transition to a 'satiated' state from which it no longer requests any candy bars. Formally, CUST-3's state has an additional 'satiated' variable besides the 'waiting' variable of CUST-1; it takes on values 'yes' or 'no', initially 'no'. CUST-3 has an additional internal action BECOME_SATIATED, defined as follows.

BECOME_SATIATED
Precondition: satiated = no
Effect: satiated := yes

Also, each of PUSH1 and PUSH2 has the additional Precondition 'satiated = no'. The BECOME_SATIATED action is in a class by itself in part(CUST-3).

Note that CUST-3 implements CUST-1, but does not solve CUST-1; there are fair behaviors of CUST-3, such as the empty sequence, that are not fair behaviors of CUST-1.

Candy Machines and Customers

Now we consider the composition of candy machines and customers. First consider the composition of CM-1 and CUST-1. Since each component preserves well-formedness, the composition has only well-formed behaviors. We claim that all fair behaviors of the composition are infinite. Suppose not: then consider any finite fair execution. By well-formedness and a simple assertion characterizing the states after finite executions, the state of the composition after the execution either has waiting = 'no' and button_pushed = 0, or has waiting = 'yes' and button_pushed = 1 or 2. In the former case, PUSH1 is enabled, while in the

latter case, either SKYBAR or HEATHBAR is enabled. But the definition of a fair execution implies that no action of the composition can be enabled in the final state.

In fact, it is not hard to see that the fair behaviors of the composition of CM-1 and CUST-1 are exactly the infinite well-formed sequences in which each dispensation action dispenses an appropriate candy (according to the most recent push).

The composition of CM-1 and CUST-2 yields exactly the sequences of the form 2,A,2,A,...,2,A,2,A..., or 2,A,2,A,...,2,A,2,H,1,S,1,S,... as its fair behaviors. The composition of CM-1 and CUST-3 produces exactly the even-length finite well-formed sequences in which each dispensation action dispenses an appropriate candy. Also, the composition of CM-2 and CUST-2 yields the single sequence 2,A,2,A,...,2,A,2,A.... as its only fair behavior. All of these, and similar characterizations for the behavior of the other compositions, can be proved by straightforward methods similar to those used above.

The previous arguments about the behavior of compositions of automata are based directly on the internal structure of the component automata. Sometimes it is possible to break up such a proof, using properties of the behavior of the component automata to prove a property of the composition. Formally, in order to prove that the composition of the automata $\{A_i\}_{i \in I}$ solves a problem, one might prove that each component automaton A_i solves a schedule module H_i, and then prove that the composition of the $\{H_i\}_{i \in I}$ solves the problem.

For example, we reconsider proving that every fair behavior of the composition of CM-1 and CUST-1 is an infinite well-formed sequence of actions in which each dispensation action dispenses an appropriate candy. Let LIVE-CUST be the schedule module whose signature is the same as CUST-1's, and whose schedules are exactly those in which 1. the customer is not the first to violate well-formedness, and 2. if the sequence is well-formed, then it is either infinite or else finite and ending with a push event. Then it is easy to see that CUST-1 solves LIVE-CUST. We have already argued that CM-1 solves the schedule module LIVE-CM described earlier. So it suffices to prove that every behavior of the composition of LIVE-CUST and LIVE-CM is an infinite well-formed sequence of actions in which each dispensation action dispenses an appropriate candy. This is not difficult to show: well-formedness holds because neither component is the first to violate it, appropriate responses follow from the specification of LIVE-CM, and the sequence is infinite because neither component stops at its own turn.

6.1.4 Choosing a Ring Leader

Now we give a brief sketch of another example, the election of a leader in a ring of processors. This example exhibits much more interesting concurrent activity than the candy machine example. It shows how one can use the model to reason about interesting concurrent algorithms, and suggests how the model can be used to carry out complexity analysis and prove lower bound and impossibility results.

We assume a ring of n processors, each starting with a unique identifier chosen from a universal totally ordered identifier set I. Each processor can communicate with each of its neighbors in the ring, using a pair of one-way channels. The processors do not know the size of the ring, nor the specific subset of I that is actually being used as identifiers. The object is for a unique processor to perform a 'leader' output action. This problem has been widely studied in the distributed algorithms research area.

Each processor and each communication channel is modelled as an I/O automaton. Each channel automaton has input actions of the form SEND(M) and output actions of the form RECEIVE(M) (note that since the model uses a global naming scheme, the actual action names would have to be subscripted with information identifying the particular channel). A channel's state is a multiset, consisting of those messages that have been sent but not yet received; initially, the multiset is empty. The transition relation is as follows:

SEND(M)
Effect: messages := messages $\cup$ {M}

RECEIVE(M)
Precondition: M $\in$ messages
Effect: messages := messages - {M}

The partition puts each different RECEIVE action in a separate equivalence class; this has the effect of hypothesizing that every message that is sent eventually gets received.

Each processor is also modelled as an I/O automaton, having SEND output actions and RECEIVE input actions. In addition, it has a LEADER output action by which it can announce that it has been chosen as the leader processor. It may also have internal actions.

A collection of channel and processor automata is composed into a single system automaton, and then the hiding operator is used to produce a new system automaton in which the only external actions are LEADER actions. The problem to be solved by the system can be described by the schedule module whose external action signature has no input actions and only LEADER output actions, and whose set of schedules consists of the set of sequences of length exactly 1. That is, in a correct behavior, exactly one LEADER event occurs.

We now describe a particular algorithm for solving this problem, based on that of LeLann [LeLann77]. Each processor sends its identifier clockwise around the ring. When a processor receives an identifier, if the identifier is less than its own, the processor discards the received identifier. If it is greater than its own, the processor passes the received identifier clockwise. If it is equal to its own, the processor performs a LEADER output action.

In more detail, the state of a processor with identifier i has a variable 'pending' which holds a subset of I, initially {i}. It also has a variable 'leader-status', which takes on values from {'unknown', 'elected', 'announced'} and has initial value 'unknown'. The steps are as follows.

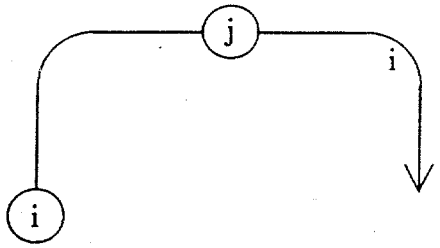


Figure 6.2: Impossible Situation if i<j

RECEIVE(j), j ∈ I
Effect: if j > i then pending := pending ∪ {j}
if j = i then leader-status := 'elected'

SEND(j), j ∈ I
Precondition: j ∈ pending
Effect: pending := pending - {j}

LEADER
Precondition: leader-status = 'elected'
Effect: leader-status := 'announced'

Each action is in a separate class of the partition. It is not hard to carry out a correctness proof of this algorithm using the model. The safety proof (i.e., that no more than one LEADER event ever occurs) involves proving an invariant assertion relating the identifiers that appear in different places in the ring, both as processor id's and in messages. More specifically, it must be shown that if i < j, then a processor with identifier i, a processor with identifier j, and a message containing identifier i cannot appear in that order, reading clockwise around the ring (see figure 6.2 below for illustration).

In order to prove liveness (i.e., that some LEADER event eventually occurs), another invariant is used, expressing conservation of the message corresponding to the maximum identifier. Then a 'variant function' is defined, describing the progress that has been made toward election of a leader: for each state, the variant function yields the remaining distance the maximum identifier needs to travel before reaching its originating processor. The value of this function is shown never to increase during execution, and at any point where it is nonzero, the fairness properties of I/O automata imply that some event will eventually occur to decrease the value. Thus, eventually, the function value reaches zero, which implies that a LEADER event occurs.

The model can be used to carry out complexity analysis. For any execution of the algorithm, the number of SEND or RECEIVE events can be used as a measure of the

amount of communication; it is not hard to see that n^2 is a worst-case upper bound on this number, where n is the number of processors in the ring. Also, for any execution, time can be measured as follows. Assign a 'real time' to each event, as large as possible, subject to the requirement that for each class of the partition, the time between successive 'turns' for that class is at most 1. Then the real time assigned to the LEADER event can be taken as a time measure for the entire execution. It is not hard to see that $2n + 1$ is a worst-case upper bound for the time measure.

The given algorithm is not optimal in its communication requirements; for example, [Peterson82] contains an algorithm with an O(n log n) upper bound. The algorithm in [Peterson82] can also be formalized and analyzed using our model. Also, [Burns80] proves an Ω(n log n) lower bound on the worst-case amount of communication; this result also is describable in our model.

6.1.5 Other Applications

The model has been used to describe and reason about many different kinds of algorithms, both in systems applications and in the algorithms research literature. In this section, we describe some of these uses.

Network Resource Allocation

Our first use of the model was for describing network resource allocation algorithms. [LynchT87] presents a network arbiter design and proves its correctness, using I/O automata. The algorithm is based on a resource performing a treewalk of a spanning tree of the network graph. The conditions proved include safety properties (mutual exclusion) and liveness properties (no lockout).

The correctness proof is done in three levels of abstraction. The problem definition is presented as a high-level schedule module, in which inputs are requests and returns, and outputs are grants, all for a particular resource. The intermediate level is a description of the algorithm in terms of graph theory, formalized as an automaton together with a restricted set of executions. Finally, the complete distributed algorithm is described as a composition of automata at the lowest level. It is shown that each level solves the level above it, and thus that the distributed algorithm solves the arbiter problem.

Most of the interesting reasoning about the algorithm is done at the intermediate level, in terms of graphs. This reasoning is close to the intuitive reasoning one would normally use to understand and explain the algorithm. The interesting work involves showing that the intermediate level solves the high-level problem statement. In contrast, showing that the lowest level solves the intermediate level is a long but straightforward case analysis.

[LynchT87] also contains an analysis of the time complexity of the algorithm, demonstrating an O(n) worst-case upper bound, where n is the number of nodes in the network,

and an O(d) worst-case upper bound when a request does not overlap with any others, where d is the diameter of the network. The time analysis proof follows the proof of 'no lockout' very closely, suggesting that there may be a general correspondence between liveness proofs and proofs of upper bounds on time.

We have also used the model to study other network resource allocation algorithms. For example, we can give an algorithm for the 'Drinking Philosophers' problem: in this problem, users request sets of resources by name, with the same user possibly requesting different sets of resources each time he makes a request. [ChandyM84] contains an algorithm for this problem, constructed by modifying a particular Dining Philosophers algorithm. Our algorithm, based on the one in [ChandyM84], is described as a composition of automata that solve the Dining Philosophers problem and automata that carry out additional bookkeeping. Our use of composition allows us to use any Dining Philosphers algorithm as a 'subroutine'; some choices can be shown to yield better time performance for the resulting Drinking Philosophers algorithm than is yielded by the algorithm of [ChandyM84].

Synchronizers

In [Awerbuch85], Awerbuch describes a *synchronizer* algorithm - a distributed algorithm designed to convert programs written for synchronous networks into versions that can be used in asynchronous networks. In this algorithm, the network nodes are partitioned into *clusters*, and different strategies are used to synchronize within clusters and among clusters. The algorithm is clever, but fairly complex, and is presented without formal proof. In [FeketeLS87], we provide a new presentation and a proof for Awerbuch's algorithm. The algorithm is decomposed into separate automata for intercluster and intracluster synchronization. The intercluster synchronizer is further decomposed into a piece executing at each node. In fact, Awerbuch's actual program for each node is described as the composition of two automata, one participating in intercluster and one in intracluster synchronization.

Communication

We have recently presented a correctness proof for the intricate distributed minimum spanning tree algorithm of [GallagerHS83]. The techniques used are based on the hierarchical structure used in [LynchT87]. However, instead of a linear hierarchy of algorithms, we use a *lattice* of algorithms. The complete algorithm has several different projections onto higher level 'subalgorithms', where each subalgorithm represents one task performed by the main algorithm. The proof involves showing that the subalgorithms all solve the minimum spanning tree problem and that the full algorithm 'solves' all of the subalgorithms. In showing the latter, we make use of many properties of the separate subalgorithms. We develop the basic theory needed for lattice-structured proofs.

More recently, we have been using I/O automata to characterize correct behavior for

physical channels and data links. We are attempting to prove that certain types of data link behavior can be implemented in terms of certain types of physical channels, while other types cannot. Preliminary results show that interesting data link behavior seems to require at least some stable storage (whereas previous work shows that a single stable bit at each end suffices). Also, it appears that the data link protocol must use unbounded size headers to achieve reasonable behavior, in case the underlying physical channels are not FIFO.

Concurrency Control

We have been using the model as the formal foundation for a new theory of *atomic transactions*. Transactions arose originally in database systems, but are now used as a basic construct for general data-oriented distributed programming. Use of transactions in general-purpose languages has required their extension to allow nesting; nested transactions permit more concurrency than single-level transactions, and permit localized handling of failures.

We can also use I/O automata to model nested transactions, state the correctness conditions that they must satisfy, describe an exclusive locking algorithm for nested transactions, and carry out a correctness proof. In later papers, we extend this treatment to more general locking algorithms and timestamp-based algorithms. We also prove correctness of algorithms for management of 'orphan' transactions - transactions that continue to execute even though some ancestor in the transaction nesting structure has been aborted. We are able to use I/O automata to decompose the orphan algorithms so that concurrency control and recovery are handled by one module, and orphan management is handled by another. Correctness properties for the two kinds of modules are proved separately, and then combined to yield correctness properties for the complete algorithm.

We have had similar success in describing correctness of algorithms for replicated data management. We are able to decompose certain replicated data algorithms into modules that handle concurrency control and recovery at the level of individual data replicas and modules that implement the data replication algorithm.

Although the model has proved to be a very usable tool for describing these results, its full power has not yet been used in this work. In particular, only finite executions have so far been considered, and only safety properties have been proved.

Shared Atomic Objects

A topic of recent research interest has been the study of wait-free implementability of concurrently-accessible atomic objects in terms of other atomic objects. An object is said to be *atomic*, roughly speaking, if it responds to concurrent invocations of operations as if the operations were executed indivisibly at some time between the invocation and response times. So far, most of the work has focussed on read-write registers for use by various numbers of readers and writers. Many of the algorithms are very complex and difficult to

understand precisely.

The paper [Lamport86], which initiated this research area, contains an interesting formal model based on partial orderings of operations. However, most of the subsequent papers do not use Lamport's model, but instead include their own models and definitions. The multiplicity of models has contributed to making the papers very difficult to read.

In [Bloom87], Bloom uses the I/O automaton model as the basis for stating correctness conditions for atomic read-write registers, for describing a new algorithm (which implements 2-writer n-reader registers from 1-writer n+1-reader registers) and for proving the algorithm correct. He describes the solution as a composition of automata for each of the reader and writer protocols and automata for the 1-writer registers used in the implementation. The combination is shown to implement the desired 2-writer register. The work is rigorous and clear; we hope that a similar presentation will help clarify some of the other algorithms.

New work by Herlihy on impossibility results for atomic object implementations also uses the I/O automaton model [Herlihy88].

Dataflow

We can formulate the semantics of dataflow networks in terms of I/O automata. We define the notion of 'determinacy' (i.e., that the sequence of output actions is uniquely defined by the sequence of input actions), a notion that is considered important in dataflow computation. We state a theorem that expresses Kahn's main result about dataflow networks - that the semantics of networks of determinate components can be uniquely defined using the least fixed point operator applied to certain equations involving behavior of the individual components. We then prove a theorem showing the equivalence of our operational semantics and Kahn's fixed-point semantics.

6.2 Exercises

1. Give a direct proof of a special case of the Burns-Lynch impossibility result, saying that 2 processes cannot achieve mutual exclusion with progress using a single read-write shared variable. Your proof should use the same basic ideas as in the n-process result, but the restriction to $n = 2$ should allow the proof to be simplified.

2. Let $\mathcal{N}$ denote the natural numbers. Consider the following two I/O automata:

   - Automaton A:
     in(A) = $\emptyset$, out(A) = {"go"}, int(A) = $\emptyset$
     states(A) = {s,t}
     start(A) = {s}

steps(A) = {(s,"go",t)}
part(A) = {{"go"}}

- Automaton B:
  in(B) = {"go"}, out(B) = {"ack"}, int(B) = {"increment"}
  states(B) = {("on",i), $i \in \mathcal{N}$} ∪ {("off",i), $i \in \mathcal{N}$}
  start(B) = {("on",0)}
  steps(B) = {(("on",i),"increment",("on",$i+1$)), $i \in \mathcal{N}$} ∪
  {(("on",i),"go",("off",i)), $i \in \mathcal{N}$} ∪
  {(("off",i),"go",("off",0)), $i \in \mathcal{N}$} ∪
  {(("off",i),"ack",("off",$i-1$)), $i \in \mathcal{N}$}
  part(B) = {{"increment"},{"ack"}}.

For each of the three automata A, B and the composition A × B, describe the sets of behaviors and fair behaviors.

3. Recall the definition of a *read-write atomic register* given in class. (Basically, all values returned are consistent with an execution in which each read and write is shrunk to an instant in the interval between the request and the acknowledgement.)

   a. Give an I/O automaton that is a read-write atomic register for 2 readers and 2 writers.

   b. Argue (briefly) that the automaton you define solves the atomic register problem in the sense that all its fair behaviors should satisfy the conditions given in class. You may first want to describe the set of allowable behaviors.

4. An I/O automaton A is *deterministic* if (a) for each state s and action π, there is at most one state s' such that $(s, \pi, s') \in$ steps(A), and (b) for each state s, there is at most one locally controlled action π for which A has a step of the form (s, π, s').

   Show that for each I/O automaton A, there exists a deterministic I/O automaton B with the same external signature as A, and such that fairbehs(B) ⊆ fairbehs(A). (It follows that B solves any problem that A does.)

6.3 Bibliography

[Awerbuch85] Awerbuch, B. Complexity of Network Synchronization. JACM 32(4), October, 1985, pp. 804-823.

[Bloom87] Bloom, B. Constructing Two-Writer Atomic Registers. *6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* Vancouver, British Columbia, Canada, August, 1987, pp. 249-259.

[Burns80] Burns, J. A Formal Model for Message Passing Systems. Technical Report TR91, Indiana University, May, 1980.

[ChandyM88] Chandy, K.M., and Misra, J. A Foundation of Parallel Program Design. Addison-Wesley, 1988.

[ChandyM81] Chandy, K.M., and Misra,J. The Drinking Philosophers Problem. ACM-TOPLAS 6(4), October, 1981, pp. 632-646.

[FeketeLS87] Fekete, A., Lynch, N., and Shrira, L. A Modular Proof of Correctness for a Network Synchronizer. *2nd International Workshop on Distributed Algorithms,* Amsterdam, The Netherlands, July,1987.

[GallagerHS83] Gallager, R., Humblet, P. and Spira, P. A Distributed Algorithm for Minimum-Weight Spanning Trees TOPLAS, Vol. 5, No. 1 (January, 1983), pp. 66-77.

[GoldmanL87] Goldman, K.J., and Lynch, N.A. Quorum Comsensus in Nested Transaction Systems. *6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,*Vancouver, British Columbia, Canada, August, 1987.

[Herlihy88] Herlihy, M. Impossibility and Universality Results for Wait-Free Synchronization. Submitted for publication.

[Hoare85] Hoare, C. A. R. Communicating Sequential Processes. Prentice-Hall, 1985.

[Kahn74] Kahn, G. The Semantics of a Simple Language For Parallel Programming. *Information Processing 74.* North-Holland Publishing Co., 1974.

[Lamport86] Lamport, L. On Interprocess Communication, Parts I and II. *Distributed Computing 1(2),* 1986, pp. 77-101.

[LamS84] Lam, S. and Shankar, U. Protocol Verification via Projections. *IEEE Trans. on Software Engineering SE10(4).* July, 1984.

[LeLann77] LeLann, G. Distibuted Systems, Towards a Formal Approach. *IFIP Congress,* Toronto, 1977, pp. 155-160.

[LynchF81] Lynch, N.A., and Fisher, M.J. On Describing the Behavior and Implementation of Distributed Systems. *Theoretical Computer Science 13,* 1981, pp. 17-43.

[LynchM86] Lynch, N.A., and Merritt, M. Introduction to the Theory of Nested Transactions. *ICDT'86 International Conference on Database Theory.* Rome, Italy, September, 1986. pp. 278-305. Also, MIT/LCS/TR-367 July 1986, to appear in *Theoretical Computer Science.*

[LynchMW] Lynch, N., Merritt, M., and Weihl, W. Atomic Transactions. In progress.

[LynchS] Lynch, N.A., and Stark, E.W A Proof of the Kahn Principle for Inout/Output Automata. Submitted for publication.

[LynchT87] Lynch, N.A., and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing.* Vancouver, British Columbia, Canada, August, 1987, pp. 137-151.

[LynchT87-2] Lynch, N.A., and Tuttle, M.R. Hierarchical Correctness Proofs for Distributed Algorithms. Master's Thesis, Massachusetts Institute of Technology, April,1987.

MIT/LCS/TR-387, April, 1987.

[LynchW] Lynch, N.A., and Welch, J.L. Synthesis of Efficient Drinking Philosophers Algorithms. In progress.

[Peterson82] Peterson, G.L. An O(nlogn) Unidirectional Algorithm for the Circular Extrema Problem. *ACM TOPLAS (4),* October,1982. pp. 758-762.

[RamadgeW85] Ramadge, P.J., and Wonham, W.M. Supervisory Control of a Class of Discrete Event Processes. University of Toronto. November, 1985. Systems Control Group Report #8515.

[WelchLL] Welch, J., Lamport, L., and Lynch, N. A Lattice-Structured Proof of a Minimum Spanning Tree Algorithm. Submitted for publication.

6.852 Distributed Algorithms	Fall Semester, 1988
Lecture 7: October 4	
Lecturer: Nancy Lynch	*Scribe: Michael Parker*

This lecture is devoted to giving examples of how to apply I/O Automata to some of the problems discussed in previous lectures.

7.1 Modeling Shared-Memory Mutual Exclusion Algorithms with I/O Automata

The I/O Automata Model was not originally designed to model shared memory algorithms. However, it is possible to do so. The following is an example of how one might model mutual exclusion algorithms based on shared registers.

Process & Shared Variable Action Signatures. Process i has actions to communicate with the outside world: try_i (input), $critical_i$ (output), $exit_i$ (input), and $remainder_i$ (output). A process i and a shared variable j have actions to communicate between each other; from the point of view of the shared variable, j has corresponding pairs of invocation and response actions $read_{i,j}$ (input) and $return_{i,j,v}$ (output) and $write_{i,j,v}$ (input) & $ack_{i,j}$ (output).[1] The variables and processes may also have an arbitrary collection of internal actions.

Process Automata Semantics. Process p_i is any arbitrary IOA satisfying the following conditions, where we assume that all states of p_i are partitioned into four classes: T, C, E, and R.

We define *externally well-formed* sequences of external actions of p_i to be those sequences β such that β restricted to outside world operations is a prefix of the infinite sequence try_i, $critical_i$, $remainder_i$, try_i,...

1. Process p_i preserves external well-formedness.
2. In any execution of p_i, whose behavior is externally well-formed, all states up to the try_i are R states, states between try_i and $critical_i$ are T states, states between $critical_i$ and $exit_i$ are C states, and those between $exit_i$ and $remainder_i$ are E states.

[1] These are simple shared variables with just read and write operations. However, the data types could be more complicated, e.g., a shared queue with insert and remove operations, etc. But no matter how complicated, they will still require an appropriate form of invocation and response actions.

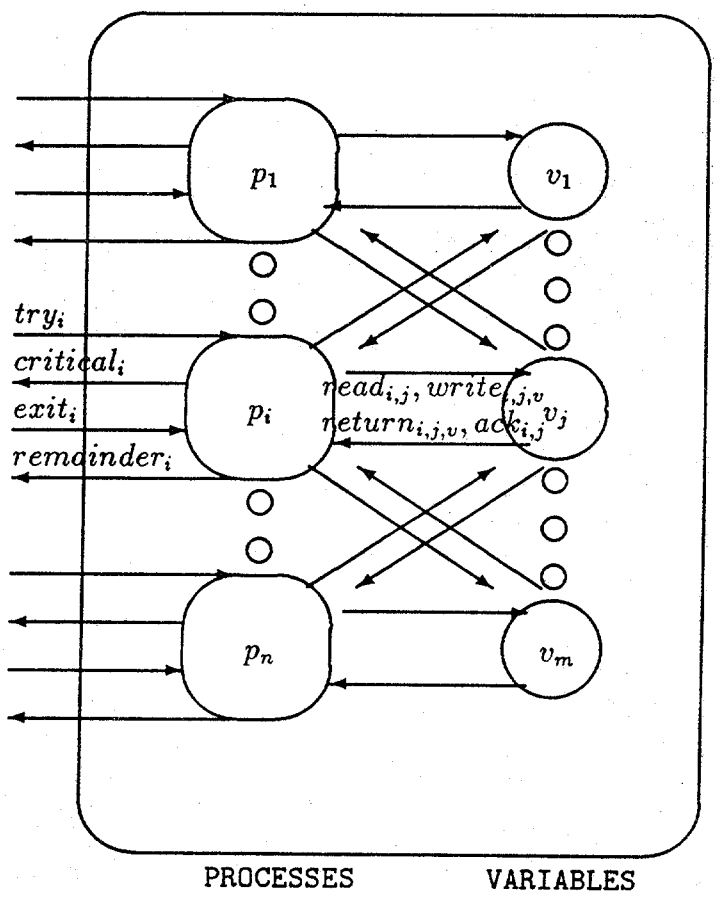


Figure 7.1: An I/O Automaton Model of a Shared-Variable Mutual Exclusion Algorithm

3. p_i's locally controlled steps are only enabled in T and E states, never in C or R.

Shared Variable Automaton Semantics For illustration, we define an atomic register. A read-write atomic register x with initial value v_0 is an IOA that solves a particular schedule module M_{x,v_0}.

We define *variable well-formed* for x and i to mean that the calls and responses involving x and i are alternating *call*, *response*, *call*,... starting with *call*, where each response corresponds to the preceding call.

Then schedule $\beta \in scheds(M_{x,v_0})$ provided that following conditions are satisfied:

1. For all i, β restricted to the actions of i is variable well-formed for x and i, or else the first violation of this property is an "incorrect call".
2. If β is variable well-formed for x and i, then each request has a corresponding later response (a liveness assumption). Moreover, it is possible to insert a $perform_{i,x}$ action between each "$request_{i,x}$" and "$respond_{i,x}$" action, in such a way that: if β is reordered with the request and respond put just before and after the perform ("shrinking the interval") and then the internal actions thrown away, the resulting new sequence γ is a correct behavior for a serial read-write object (each read gets the value of the preceding write, if any, or v_0 if none). It is "as if" each operation occurred at some time during the given interval.

By imposing these restrictions, we can model an atomic object with non-atomic operations. Of course, we can also generalize this definition to more general data types, including general *test & set*.

7.2 Modeling Safe Registers with I/O Automata

For the special case of a read-write variables with 1 writer, we can define a weaker kind of register called a "safe" register. For a safe register, recall that reads not overlapping any writes get the last preceding write value, while reads overlapping writes get arbitrary values.

The definition uses the first condition of M_{x,v_0} for well-formedness, but now Condition 2 is modified. The definition is as follows.

Sequence $\beta \in scheds(M_{x,v_0})$ provided that following conditions are satisfied:

1. For all i, β restricted to the actions of i is variable well-formed for x and i, or else the first violation of this property is an "incorrect call".
2. If β is variable well-formed for x and i, then each request has a corresponding later response (a liveness assumption). If a read request overlaps a write request, an arbitrary value is returned.

7.3 Mutual Exclusion Correctness Conditions using I/O Automata

Formally, the mutual exclusion problem can be expressed in terms of solving a schedule module M with.

Signature For each process i, $acts(M)$ contains:

INPUTS: try_i, $exit_i$, and

OUTPUTS: $critical_i$, $remainder_i$.

Schedules A sequence β of actions of M is in $scheds(M)$ iff the following conditions hold:

1. For all i, β preserves external well-formedness for i.
2. (Mutual Exclusion) If β is externally well-formed for all i, then there is no point in β in which two processes are in the *critical* region, where being in the critical region means that $critical_i$ was the most recent action of i.
3. (Progress) If β is externally well-formed for all i, and each $critical_i$ has a later $exit_i$, then for every point in β at which someone is not in the *remainder* region, there exists a later $critical_i$ or $remainder_i$ action.[2]

It is instructive to reformulate our earlier notions of "normal operation" and "normal progress" using this model:

Normal Operation is captured by

- The fairness definition for the process I/O Automata,
- The schedule module definitions for the shared variables that say each request gets a later response, and
- The hypothesis in the conditional problem specification saying that each $critical_i$ is eventually followed by $exit_i$.

Normal Progress is given in the definition of M.

[2]Or, you could say the sequence cannot be finite and end with anything other than all processes in the *remainder* region. These statements are equivalent in the sense that a solution to one is a solution to the other (Exercise: Prove this).

A difference between this model and the original informal model is that this one has atomic registers with separate actions, while the other had requests and responses as one atomic action. The difference is significant if we want to carry out formal correctness proofs, because the extra actions add to the number of interleavings we need to consider, and therefore add to the complexity of the proofs. For example, an invariant assertional proof would have more values of the program counter. Luckily, there is a theorem that will allow us to consider only executions in which requests and responses are atomic:

Theorem 7.1 *Let S be a system of processes and shared atomic variables. Suppose (technical assumption) in all executions of each p_i whose behaviors are externally well-formed, if p_i invokes an operation on a variable, then no locally controlled action of p_i occurs until a corresponding response happens. Let α be a fair execution of S whose behavior is externally well-formed for each i. Then there exists a fair execution α' of S such that:*

1. *In α', the only occurrence of request and response actions occur as pairs of corresponding actions.*
2. *$beh(\alpha) = beh(\alpha')$.*

Proof: Start with α and do some reordering to get α'. First get "perform" actions inserted (by definition of atomic objects). Then move the request and response actions to bracket the performs. We can do this without moving any actions of a process p_i past any other action of p_i[3]. Now we remove the performs and adjust the states accordingly to reflect the moved actions. The result, α', is also a fair execution of S whose behavior is externally well-formed for each i. That the two required properties hold for α' is obvious. ∎

This theorem can be used to simplify correctness proofs. Consider the mutual exclusion correctness conditions given earlier. To apply this theorem we still need to show Condition 1 directly (that α preserves external well-formedness). But the other two conditions, mutual exclusion and progress, are conditions on the external behavior of fair executions with externally well-formed behaviors. By this theorem, if something is true for all fair executions in the given restricted form, it's true for all fair executions. So it suffices to consider fair executions in the restricted form only.

This theorem can be used to carry out invariant assertional proofs. For example, one can carry out an assertional proof of mutual exclusion. Using the theorem, consider the fair executions of S that have corresponding requests and responses consecutive. We can consider the system states before and after such pairs in stating the invariant, but ignoring the intermediate states. The system states are similar to those we informally described earlier:

[3]This is because of the technical assumption in the theorem that says p_i doesn't do anything (locally controlled) from the time when it requests an operation and until the time when it gets a response. Also, since p_i invoked an operation, it must be in the trying or critical regions by basic process assumption. Then since p_i is externally well-formed, no input actions come into p_i from the outside.

System State = process states + variable states (states of IOA's).

Process States are the IOA state, summarizing the program counter, local variables, region designation, etc., all included in one state.

Variable States are a bit of a problem. Registers are defined to be *arbitrary* IOA's subject to certain correctness conditions. So we don't really know what the states are!

But there aren't a lot of choices: the behaviors of an atomic register that arise from fair executions with the pairing property just look like $request, response, request, response, \ldots$, where (by the shrinking property), these must give responses as in a serial object. So we might as well assume we have a serial object – after each pair, the state is updated as in usual sequential programming languages. For example, for a read-write variable,

- the state starts at an initial state,
- after a *read*, the state is unchanged, and
- after a *write*, the new state is the value written.

So we can regard the state as just the data value. This leads to proofs like the ones we did earlier.

7.4 Time Bounds

How do we do time analysis like the Peterson tournament analysis, using IOA's? The I/O Automaton model per se does not provide any direct support for such analysis; there's no notion of time defined formally. Further research on using I/O Automata models for time analysis is needed. As a possible direction, suppose we add time information to executions in order to analyze them. We define a *timed execution* to be an execution in which each event has an associated non-negative number such that the sequence of those numbers, ordered according to the position of the events in the execution, is monotone non-decreasing. The numbers indicate the time at which the corresponding events occur.

We need some assumptions to give upper bounds on the difference between times for certain events. For example, we might like to bound the time between $critical_i$ and $exit_i$ to a constant b, or to bound by a constant the time from any point in an execution until the next time when either a locally controlled action of p_i occurs or none is enabled.

This latter condition seems like it would fit nicely into the definition of an IOA. We could define an augmented IOA called a *timed I/O Automaton* whose executions are defined to be timed executions. Instead of fairness, where each class in the partition gets a turn infinitely often, we use a notion of "time-bounded fairness": we associate a t_c to each class C and

say each class gets a turn within time t_c. (Formally, t_c is the upper-bound on the difference between the times when C takes steps or has no action enabled.)

By specifying the time bounds on the externally controlled events (in the case of the shared registers, the time to go from the *critical* region to the *exit* region), then we can derive the time bounds on the remaining events, such as an upper bound on the time in the trying region.

7.5 Deterministic verses Non-Deterministic Algorithms

Recall that in the impossibility proofs, we assumed a deterministic algorithm. When p_i took a locally controlled step, which action it took and which state resulted was determined. And when an input action occurred for p_i, the next state was determined.

Question: Why does proving impossibility for determined algorithms to solve a problem imply impossibility for arbitrary algorithms to do so?

The answer is part of a general result that says:

Theorem 7.2 *For an I/O Automaton A, there exists a deterministic I/O Automata A' with $fairbehs(A') \subseteq fairbehs(A)$.*

This theorem says if A solves a problem, then there is a deterministic I/O Automata which also solves the problem. So conversely, impossibility for deterministic automata implies impossibility for all automata. The proof of the theorem is left as an exercise.

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 8: October 6

Lecturer: Nancy Lynch *Scribe: Mark Tuttle*

Until this point in the course, all of the algorithms we have studied have been *deterministic algorithms*, algorithms in which the current local state of a processor (possibly together with the shared memory) uniquely determines the next action the processor will perform, and hence uniquely determines the processor's next state. This lecture introduces a new class of algorithms called *randomized algorithms*, in which the current local state of a processor determines only a *set* of possible next states, and in which a processor chooses its next state by selecting one of these possible next states at random according to some probability distribution. Most frequently a processor's current state determines a set of only two possible next states and the processor chooses its next state by flipping a fair coin, choosing one state if the outcome of the coin toss is heads and choosing the other if the outcome is tails. This simple but surprisingly powerful idea of allowing processors to flip coins during computation was first introduced to distributed computing by Michael Rabin. In this lecture, we study his randomized algorithm for mutual exclusion [Rabin82].

8.1 Randomized Algorithms

Whenever we prove a lower bound for a problem in this class, we do so by first assuming the existence of a solution consuming less of a resource than is demanded by the lower bound, and then constructing a strange execution of this algorithm that violates one of the problem's correctness conditions; for example, we might construct an execution in which a processor takes its steps only when certain shared variables are set to certain tricky values that keep the processor from making any progress during the execution. The construction of such an execution is facilitated by the fact that a processor's next state is completely determined by its local state and the shared memory: we show that, starting from a given global state, it is possible to schedule processor steps in such a way that the system must return to this global state, resulting in an undesirable infinite looping behavior. Sometimes, however, this cycle can be broken if we allow processors to flip coins as part of their computation. This is one powerful feature of randomized algorithms. Furthermore, it is natural to believe that these strange executions constructed during lower bound proofs are extremely unlikely to occur in practice, and on many occasions we are willing to accept an algorithm that makes mistakes as long as it behaves correctly "most of the time." A second powerful feature of randomized algorithms is that by allowing processors to flip coins during computation we are able to impose a natural probability distribution on the set of executions of the algorithm and make formally precise this idea that the algorithm behaves correctly "most of the time." There

are, for example, randomized algorithms for mutual exclusion that on rare occasions violate the no lockout condition, but for which we can prove that "with probability 1, any processor in its trying region will eventually enter its critical region."

Making probabilistic statements about randomized algorithms, however, is often very difficult. This difficulty has two aspects: the first difficulty is the definition of the probability space underlying the statement, and the second difficulty is the formulation of the statement itself.

Probabilistic statements like "the probability that a toss of a fair coin results in heads is $\frac{1}{2}$" only make sense in the context of a probability space. A probability space consists of a sample space and a probability measure assigning probabilities to various subsets of the space. For example, in the preceding statement, the sample space might be the set $\{H, T\}$ consisting of the two outcomes of the coin toss, and the probability measure might assign probability $\frac{1}{2}$ to each event $\{H\}$ and $\{T\}$ that the outcome of the toss is heads or tails, respectively. Since the probability space underlying a probabilistic statement is usually clear from context, we are often quite sloppy about defining the probability space. When making probabilistic statements about randomized algorithms, however, subtle changes in the probability space can often lead to important differences in the meaning of the statement, so it is important to define the probability spaces carefully.

Let us consider the definition of a probability space acceptable for use when making probabilistic statements about a randomized algorithm. Suppose we take as the sample space the set of all possible executions of the algorithm, and let us consider the problem of defining an appropriate probability measure on this set of executions. Given a particular execution, how do we determine its probability of occurring? Notice that this execution is uniquely determined by two things: the schedule of processor steps occurring during the execution, and the sequence of coins flipped by the processors during the execution. Notice also that if we assign to every execution a probability of occurring, then conditional probability induces a natural probability distribution on the set of possible schedules and on the set of possible coin flip sequences.

While it is natural to assign a probability to a sequence of coin flips ("the probability of two heads in a row is $\frac{1}{4}$"), is it natural to assign a probability to a schedule of processor steps? We often think of the choice of the next processor to take its step as a nondeterministic choice, but it might initially seem natural to assume that all schedules are equally likely. Unfortunately this may result in a statement that is too weak to be of any general interest: since an operating system may use a particular scheduling algorithm that may guarantee that certain schedules will never occur, a result describing the behavior of the algorithm when all schedules are equally likely will give us no information about the behavior of the algorithm running under such an operating system.

To avoid the problem of assigning probabilities to schedules, we often assume that the schedule is under the control of an *adversary* that determines the order in which processors take steps. In general, we let the adversary control those factors (such as the processors'

initial input) that can influence an execution but to which we do not want to assign a probability distribution. The execution of the protocol is an interaction (a game) between the processors and the adversary. We will see below why this is a solution to our problem.

Notice that this solution has its own headaches, since now we must formally define what an adversary is. What information is the adversary allowed to use when it chooses the next processor to take its step: can it use both a processor's local state and the shared memory, can it use only the shared memory, can it use the sequence of coins processors have flipped so far, can it use only the "success" or "failure" of a processor's last attempt to make progress (e.g., whether or not it entered the critical region), etc.? When is the adversary allowed to choose the next processor to take its step: can it make the choice interactively during the execution, or must it choose the entire schedule at the beginning of the execution before any processor has taken its first step (the first is clearly a more powerful form of adversary than the second)? Can the adversary be viewed as a deterministic algorithm, a probabilistic algorithm, a nondeterministic algorithm, etc.? All of these question must be answered.

Let us suppose we have answered these questions, and let us see why the definition of an adversary is useful. Notice that if the adversary is a deterministic algorithm, then an execution of a protocol P under the control of an adversary A is determined uniquely by the sequence of coins flipped during the execution, and we know how to assign probabilities to sequences of coin flips. Thus, having defined an adversary, instead of making statements like "condition C holds with probability 1" (where, as noted above, the underlying probability distribution on executions must implicitly impose a probability distribution on the set of possible adversaries), we make statements like "for every adversary A, condition C holds with probability 1." What is the probability distribution underlying such a statement? Notice that if we define for every adversary A a sample space consisting of all executions of protocol P under the control of adversary A, then every execution in the space is uniquely determined by the coin flip sequence appearing in the execution. It is very natural to define a probability measure on this space that assigns to every set of executions the probability of the coin flip sequences that appear in the executions in this set of executions. Denoting the resulting probability space by $\mathcal{P}(A)$, the statement "for every adversary A, condition C holds with probability 1" means that "for every adversary A, condition C holds with probability 1 in $\mathcal{P}(A)$."

In addition to the definition of the underlying probability space, we also mentioned that a second difficulty with making probabilistic statements about randomized algorithms is in the formulation of the statement itself. This is usually due to the ambiguity of the English language. For example, the statements "with probability p, if condition C_1 holds then condition C_2 holds" and "if condition C_1 holds, then with probability 1 condition C_2 holds" could be interpreted as having very different meanings. Recall that the implication "$X \supset Y$" means "either $\neg X$ holds or Y holds." The first condition, therefore, could be true even though C_2 is *always* false simply because $\neg C_1$ holds with overwhelming probability (e.g., with probability at least p). The second condition seems to be say that for a fraction p

of the times C_1 holds, C_2 holds as well (i.e., it could be interpreted as a statement about conditional probability). Finally one point of confusion for those unfamiliar with probability is that "with probability 1" does not necessarily mean "with certainty." Consider the game in which a player tosses a fair coin repeatedly and wins as soon as one of the tosses results in heads. Notice that the player wins in every play of the game except on that rare occasion when the player tosses nothing but tails forever. This happens, of course, with probability 0. The player therefore wins the game with probability 1 even though there is one rare instance in which the player loses.

8.2 Rabin's Mutual Exclusion Algorithm

Having come to terms with some of the subtleties of randomized algorithms, let us take a look at Michael Rabin's randomized algorithm for mutual exclusion [Rabin82]. This algorithm solves a probabilistic version of the no-lockout mutual exclusion problem using a test-and-set primitive on a shared variable with $O(\log n)$ values. Recall that [BurnsJLFP82] proves a lower bound of $\Omega(n)$ on the number of values the shared variable must assume to solve the same problem deterministically. Rabin's algorithm is an example of how randomized algorithms are sometimes provably better than deterministic algorithms. Rabin's algorithm is also an example of how randomized algorithms are often simpler to state than deterministic algorithms for the same problem, although their analysis can be much more difficult.

The basic idea of Rabin's algorithm is that each round of competition for entry to the critical region consists of a lottery in which each processor draws a number at random, and the processor drawing the largest number is allowed to entry the critical region. The algorithm uses a test-and-set primitive on a shared variable $V = (S, B, R)$ with three fields:

1. $S \in \{0, 1\}$ is used as a semaphore to ensure mutual exclusion as in simple mutual exclusion algorithms where a processor enters the critical region only when the semaphore is set to 0, sets the semaphore to 1 when entering the critical region, and resets the semaphore to 0 when leaving;

2. $B \in \{0, \ldots, b\}$ is used to post the largest number drawn by a processor in the current lottery for entrance to the critical region; and

3. $R \in \{0, \ldots, r\}$ is used to post a round number for the current round of competition for entry to the critical region.

To establish the $O(\log n)$ bound on the number of values assumed by V, Rabin takes b to be $\lceil \log n \rceil + 4$ and r to be a small constant such as 99.

The algorithm itself appears in Figure 8.1. To enter the critical region, processor p_i chooses a lottery number b_i from $\{1, 2, \ldots, b\}$ at random according to the probability distri-

Shared variable: $V = (S, B, R)$, where

$S \in \{0, 1\}$, initially 0 ** semaphore **
$B \in \{0, 1, \ldots, \lceil \log n \rceil + 4\}$, initially 0 ** posted number **
$R \in \{0, 1, \ldots, 99\}$, initially 0 ** round number **

Local variables: $\forall i$,

B_i is the chosen number of p_i, initially 0
R_i is the round number of lottery in which p_i is currently participating, initially $\perp$

Code for p_i:

```
while V ≠ (0, B_i, R_i) do        ** if not winner at same time C is available **
  if (V.R ≠ R_i) or (V.B < B_i) then     ** not yet participated in lottery **
    B_i ← random
    V.B ← max(V.B, B_i)
    R_i ← V.R
  unlock; lock;
V ← (1, 0, random)
unlock;

 ** Critical Region **

lock;
V.S ← 0
R_i ← ⊥
B_i ← 0
unlock;

 ** Remainder Region **

lock;
```

Figure 8.1: Rabin's randomized mutual exclusion algorithm.

bution

$$\Pr[b_i = j] = \begin{cases} (\frac{1}{2})^j & \text{if } 1 \leq j \leq b-1 \\ (\frac{1}{2})^{b-1} & \text{if } j = b. \end{cases}$$

Equivalently, p_i chooses b_i by repeatedly flipping a fair coin, counting the number of times the coin is flipped until it comes up heads, and setting b_i to the number of the flip that came up heads. Since, b_i must assume values in $\{1, \ldots, b\}$, p_i sets b_i to b if the coin does not come up heads within b flips (that is, if the coin comes up tails b flips in a row). Having chosen its lottery number b_i, p_i updates the maximum lottery number chosen by a processor during the current round, a value posted in B, by setting B to the maximum of b_i and B. The processor then determines whether it has won the lottery: if the mutual exclusion semaphore S is set to 0 and p_i's lottery number b_i is the maximum lottery number B chosen during the current round, p_i sets the semaphore S to 1 and enters the critical region. For the sake of other processors competing for the chance be the next to enter the critical region, p_i resets the maximum lottery number B to 0 before entering. Upon leaving the critical region, p_i resets the mutual exclusion semaphore S to 0 to allow other processors to enter. This is the essence of the algorithm. Since, however, p_i should play the lottery at most once per round, a round number for each round of competition is posted in R to help processors distinguish between rounds. The round number for any given round is chosen a random from $\{0, \ldots, r\}$ by the winner of the previous round as it enters the critical region. Processor p_i records the current round number R in r_i when it chooses b_i. Suppose p_i has chosen a ticket during the current round (that is, p_i chooses b_i after the last processor to enter the critical region has done so). Then since B and R are reset only when a processor enters the critical region, we must have (i) $b_i \leq B$, since p_i sets B to the maximum of b_i and B when it chooses b_i, and (ii) $r_i = R$, since p_i sets r_i to the value of R when it chooses b_i. Since p_i should play the lottery at most once per round, p_i checks that at least one of these conditions is false before choosing a new lottery number. To increase its confidence that it has actually won the current lottery (and not a past lottery) when it enters the critical region, p_i checks that $R = r_i$ before entering the critical region.

So far we have described a solution to a problem we have not defined. It is easy to see that Rabin's algorithm satisfies mutual exclusion since we are using a semaphore to guard access to the critical region, and that Rabin's algorithm satisfies no deadlock since some processor wins every round and is able to proceed to the critical region. Neither of these statements are probabilistic statements. The algorithm also satisfies a probabilistic version of no lockout saying roughly that if a processor in its trying region participates in round k, then with probability at least c/n it will enter its critical region in round k (here c is some constant).

Before making this statement precise, however, we must define the adversary (in a way that depends only on our model of computation, and not on Rabin's particular mutual exclusion algorithm). In our case, the adversary will be required to choose the next processor allowed to take a step as a deterministic function of the sequence of past processors steps and re-

gion changes. A *run* is a finite or infinite sequence of the form $i_1(old_1, new_1), i_2(old_2, new_2), \ldots$ where i_j denotes an index of a processor p_{i_j} and (old_j, new_j) denotes a region change $old_j \rightarrow new_j$. Such a run is said to be the run of an execution if i_j denotes the processor p_{i_j} taking the jth step in the execution and (old_j, new_j) denotes the region change $old_j \rightarrow new_j$ processor p_{i_j} underwent during this step in the execution. The run of a finite prefix of an execution is defined analogously. If a run r is the run of an execution e or if r is the run of a finite prefix of e, then we say that e is *compatible* with r. Notice that there may be many executions compatible with a given run. An *adversary* is a mapping A from the set of finite runs to the set $\{1, \ldots, n\}$ determining what processor is to take its next step as a function of the current prefix of the current run. A run $i_1(old_1, new_1), i_2(old_2, new_2), \ldots$ is said to be *compatible* with an adversary A if $A[i_1(old_1, new_1), \ldots, i_j(old_j, new_j)] = i_{j+1}$ for every j. An adversary A is said to be *normal* if for every infinite run compatible with A and every processor p_j the following condition holds: if the last occurrence of j appears in the run as i_k, then $new_k = R$ (that is, if p_j takes only a finite number of steps, then its last step leaves it in the remainder region).

As previously mentioned, the probabilistic version of the no lockout condition satisfied by Rabin's algorithm essentially says that if processor p_i participates in round k, then p_i enters the critical region in round k with probability at least c/n. This makes some sense since each processor playing the lottery is equally likely to win, and hence each processor playing the lottery should have roughly $1/n$ probability of winning and entering the critical region. To make this statement precise, we must provide a definition of participation and of a round. Again, these definitions must depend only on the model of computation, and not on Rabin's mutual exclusion algorithm. We define a round of an execution to be a sequence of processor steps from the time one processor enters its critical region until the time the next processor enters its critical region; formally, a *round* of an execution is a maximal subsequence of the execution containing one transition $T \rightarrow C$ at the end of the subsequence and containing no other transition $T \rightarrow C$. We say that a processor p_i *participates* in a round if a transition $T \rightarrow C$ or $R \rightarrow T$ or $T \rightarrow T$ by p_i appears in that round. The precise statement of the no lockout condition is as follows:

Theorem 8.1 *For every normal adversary A and every k-round run α compatible with A, the probability that p_i enters the critical region in round $k+1$, given that p_i participates in round $k+1$ of an execution compatible with α, is at least c/n for some constant c.*

Let us reiterate the meaning of this statement: Choose a normal adversary A, and consider the probability space $\mathcal{P}(A)$ of all runs of Rabin's algorithm with the adversary A. Let $E(p_i, k+1)$ be the set of executions in which processor p_i enters the critical region in round $k+1$, and let $P(\alpha, p_i, k+1)$ be the set of all executions compatible with α in which p_i participates in round $k+1$. For every k, every k-round run α compatible with A, and every p_i, $\Pr[E(p_i, k+1) \mid P(\alpha, p_i, k+1)] \geq c/n$.

Proof: The proof of Theorem 8.1 follows from three lemmas.

Lemma 8.2 *Let each of n players independently flip fair coins, let k_i be the first of player i's flips coming up heads, and let $k = \max\{k_i : i = 1, \ldots, n\}$. The probability that $k \leq b = \lceil \log n \rceil + 4$ is at least $(\frac{1}{e})^{\frac{1}{16}} \approx .94$.*

Proof: This follows by a simple probability argument; see [Rabin82] for details. ∎

Lemma 8.3 *Let each of n players independently flip fair coins, and let k_i be the first of player i's flips coming up heads, and let $k = \max\{k_i : i = 1, \ldots, n\}$. The probability that two of the k_i are equal to k is at most $\frac{1}{3}$.*

Proof: This follows by a longer, but standard, argument; see [Rabin82] for details. ∎

Lemma 8.4 *Assume that each of $m \leq n$ processors p_i choose lottery numbers b_i as required by the algorithm. The probability that two processors p_j and p_k choose numbers b_j and b_k equal to $\max\{b_i : i = 1, \ldots, m\}$ is at most $\frac{1}{3} + .06$.*

Proof: The probability that two processors choose the maximum lottery number is equal to the sum of

1. the probability that two processors choose the maximum number and in the process flipped no more than $b - 1$ tails, and
2. the probability that two processors choose the maximum number and in the process flipped b tails.

The probability 1 is bounded above by the probability two players draw the maximum in the infinite game described in Lemma 8.3, which is at most $\frac{1}{3}$. The probability 2 is bounded above by the probability that the maximum in the infinite game described in Lemma 8.2 is greater than b, which is at most $1 - .94 = .06$. ∎

We now finish the proof of Theorem 8.1. Consider the processor p_i that we assume participates in the $k+1$st round. (For notational convenience, let us assume that all probabilistic statements in the remainder of this proof are implicitly conditioned on the event that p_i participates in the $k+1$st round of an execution compatible with α.) We have previously argued (informally) that a processor plays the lottery at most once in any round, but notice that it is possible for p_i to participate the $k+1$st round and yet not play the lottery: it is possible that when p_i last played the lottery the round number was ℓ, and that the last processor to enter the critical region drew ℓ when it drew the round number for the $k+1$st round, in which case p_i may be fooled into not playing the lottery during the $k+1$st round since $R = r_i$. Since, however, the round number was chosen at random from the set $\{0, \ldots, 99\}$ (recall that $r = 99$), p_i is fooled with probability at most .01. Thus with probability at least

.99 processor p_i sees $R \neq r_i$ and plays the lottery during the $k+1$st round. It follows that (conditioning on whether p_i plays the lottery)

$$\begin{aligned}\Pr[p_i \text{ enters C during round } k+1] &\geq \Pr[p_i \text{ enters C} \mid p_i \text{ plays}]\Pr[p_i \text{ plays}] \\ &= .99\Pr[p_i \text{ enters C} \mid p_i \text{ plays}].\end{aligned}$$

We must compute $\Pr[p_i \text{ enters C} \mid p_i \text{ plays}]$.

Consider the set S of processors that, at the end of the $k+1$st round, have recorded the current round number R as the round number. These are the only processors that can possibly enter the critical region at the end of the $k+1$st round. Notice that if p_i is the only one of these processors to draw the winning number (the maximum drawn by processors playing the lottery during this round) then p_i will enter the critical region at the end of the round. Thus, $\Pr[p_i \text{ enters C} \mid p_i \text{ plays}] \geq \Pr[p_i \text{ alone draws max} \mid p_i \text{ plays}]$.

The computation of $\Pr[p_i \text{ alone draws max} \mid p_i \text{ plays}]$ follows from Lemma 8.4, modulo one technical point. Some of the processors in S actually play the lottery during the $k+1$st round, and some of the processors have played and lost the lottery during previous rounds (rounds in which the round number was also R). Since these processors have lost previous rounds, intuitively their numbers must tend to be rather small. That is, the probability distribution for their choice of numbers should be skewed to smaller numbers than the probability distribution for the choice of processors who actually play in the $k+1$st round. This should, intuitively, increase the probability that p_i (who actually plays in this round) is the processor choosing the *maximum* number. Since all we want to do is prove a lower bound on the probability p_i chooses the maximum uniquely, it is acceptable to assume in our analysis that all processors in S actually play the lottery during the $k+1$st round; the result will be slightly lower than p_i's actually chance of choosing the maximum. We are handwaving here, but to the best of our knowledge, no one has produced a more rigorous analysis; and coming up with a better analysis is an interesting problem to work on. Making this assumption, Lemma 8.4 tells us that the probability the maximum is chosen uniquely is $2/3$, and since all processors in S are equally likely to choose the maximum, the probability p_i is the one that chooses this maximum is $1/|S| \geq 1/n$. Thus,

$$\Pr[p_i \text{ alone draws max} \mid p_i \text{ plays}] \geq \frac{2}{3n}.$$

It follows that $\Pr[p_i \text{ enters C during round } k+1] \geq .99\frac{2}{3n} = c/n$ for some $c \approx 2/3$. ∎

We note that it is possible to extend Rabin's result and prove the following:

Corollary 8.5 *Let A be a normal adversary, and let α be a k-round run compatible with A at the end of which processor p_i is in the trying region. The probability p_i enters the critical region within the first ℓ rounds in which it participates following the kth round, given an execution compatible with α, is at least $1-(1-c/n)^{\ell}$.*

Proof: It is not hard to use Theorem 8.1 to prove that the probability p_i *fails* to enter the critical region in the first round in which it participates after the kth round, given an execution compatible with α, is at most $1 - c/n$. Proceeding by induction on ℓ, it is not hard to use this fact to prove that the probability p_i fails to enter the critical region within the first ℓ rounds in which it participates after the kth round, given an execution compatible with α, is at most $(1 - c/n)^\ell$. Consequently, the probability of *success* is at least $1 - (1 - c/n)^\ell$. ■

As $1 - (1 - c/n)^\ell$ tends to 1 as ℓ tends to infinity, we have the following:

Theorem 8.6 *Rabin's algorithm satisfies the following correctness conditions:*

1. *mutual exclusion: at most one processor is in the critical region at any time.*
2. *no deadlock: if at some point some processor is in the trying region, then some processor eventually enters the critical region.*
3. *no lockout: for every normal adversary A, with probability 1, if at the end of round k processor p_i is in the trying region, then in some later round processor p_i enters the critical region.*

8.3 Ben-Or's Mutual Exclusion Algorithm

While randomized mutual exclusion using a test-and-set primitive on a shared variable with $O(\log n)$ values is an improvement over deterministic mutual exclusion using $\Omega(n)$ values. Ben-Or has actually come up with a mutual exclusion algorithm satisfying the conditions listed in Theorem 8.6 using only a *constant* number of values. Like Rabin's algorithm, Ben-Or's algorithm uses a shared variable $V = (S, B, R)$ where S, B, and R are all 0/1 variables. The difference between Rabin's and Ben-Or's algorithm is that the lottery number b_i is chosen to be 0 with probability $1 - 1/n$ and 1 with probability $1/n$.

6.852 Distributed Algorithms — Fall Semester, 1988

Lecture 9: October 13

Lecturer: Nancy Lynch — *Scribe: Boaz Ben-Zvi*

9.1 Mutual Exclusion in Distributed Networks

In the previous lectures we used shared memory as a communication model for processes. Now lets consider a different architecture – a *network*, and let processes communicate by *message-passing*.

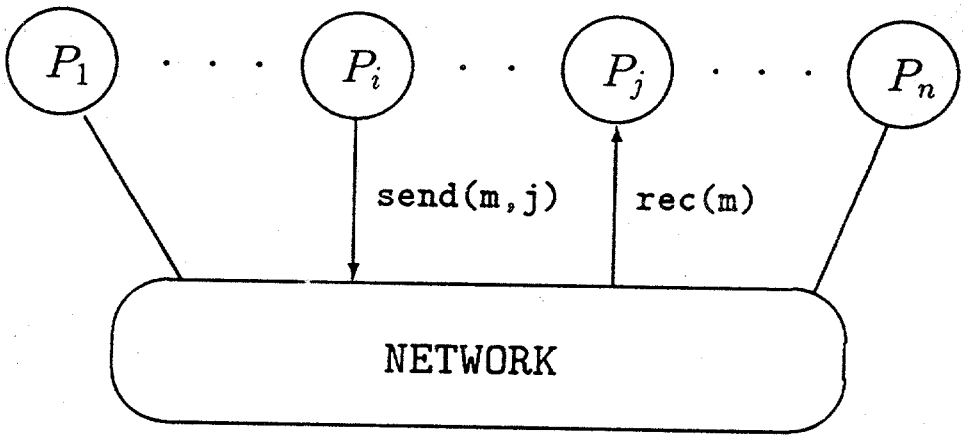


Figure 9.1: Distributed Network: processes communicating by message-passing.

When a process P_i does *send(m,j)*, it sends a message m to process P_j. The network guarantees that the message will eventually arrive at P_j. A process can also do *broadcast(m)* meaning: "do *send(m,j)* to all j".

We would like to implement mutual exclusion in this model. A straightforward solution is to simulate single-writer shared variables by making them internal to processes. A process can write to its internal variable, and others can read it by sending messages. This solution may require a large number of messages and not be very efficient. Many messages may be used to read a variable, returning the same value, until it is changed. Another idea is to send messages only when values change. A process that received no message can assume that the variable's value has not changed. This idea can lead to some specially tailored solutions.

9.1.1 Modeling

In Figure 9.2, we model a node as two I/O Automata. The buffer (network) is also described as an I/O Automaton.

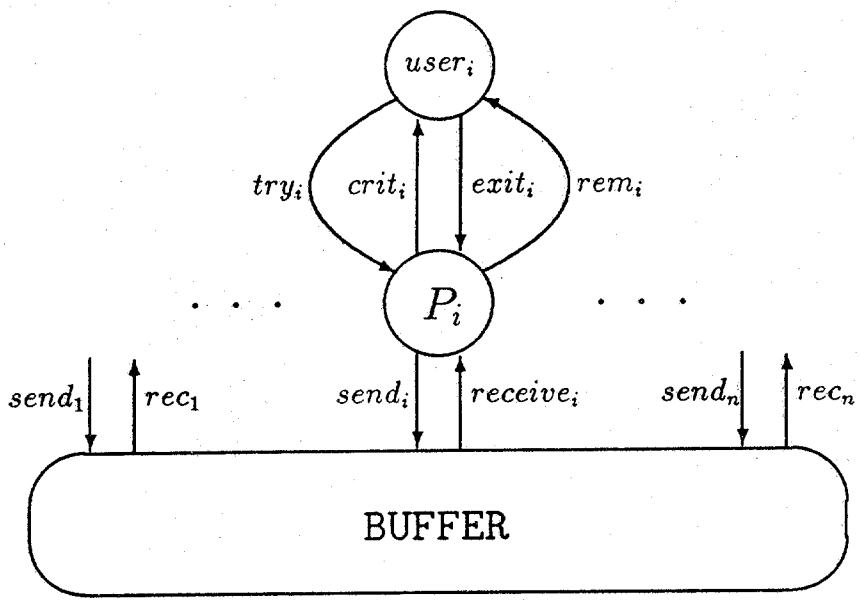


Figure 9.2: Distributed network modeled as I/O Automata.

The conditions for normal operation, in well-formed fair behaviors are:

- The $user_i$ IOA must guarantee that any $crit_i$ action is eventually followed by an $exit_i$ action. (That is, every user entering the critical region eventually exits it.)
- The *BUFFER* IOA must guarantee that any *send(m,j)* action is eventually followed by a *rec(m)* action. (That is, every message sent is eventually delivered.)
- The P_i IOA continues to take steps (by fairness definition of IOA). In this model it is allowed to take steps in all regions, including the Critical and the Remainder regions.

9.1.2 Le Lann 1977

Le Lann proposed a simple solution: the processes are to be arranged in a logical ring $P_1 \rightarrow P_2 \rightarrow \cdots \rightarrow P_n \rightarrow P_1$. A *token* that represents the resource is passed around the ring in order. When a process P_i receives a token, it checks for an outstanding request for the resource from $user_i$. If there is no such request, the token is passed to the next process

in the ring. If there is an outstanding request, the resource is granted and the token is held until the resource is returned and then passed to the next process.

Code for P_i:

local variables:		$token \in \{none, available, in\_use, used\}$
		$region \in \{R, T, C, E\}$
initial state:		$token = available$ at P_1, $none$ elsewhere.
		$region = R$
try_i	no preconditions.	
	effect:	$region \leftarrow T$
$crit_i$	precond:	$region = T$, $token = avail$
	effect:	$region \leftarrow C$, $token \leftarrow in\_use$
$exit_i$	no preconditions.	
	effect:	$region \leftarrow E$
rem_i	precond:	$region = E$
	effect:	$region \leftarrow R$, $token \leftarrow used$
$receive(t)$	no preconditions.	
	effect:	$token \leftarrow available$
$send(t, i+1)$	precond:	$token = used \ \vee \ (token = available \ \wedge \ region \neq T)$
	effect:	$token \leftarrow none$

Properties:

- *Mutual Exclusion:* exists in normal operation because there is only one token, and only its holder can have the resource.
- *Progress:* exists in normal operation because the process who holds the token is either:
  - in C: then eventually will go to E.
  - in T: then can go to C.
  - or in E or R: then has to pass the token to the next process.
- *Fairness:* exists in normal operation because a process with a request has to wait for less than n others.
- *Resiliency:* (discussed in the Le Lann paper)
  - *Process failure:* When a process fails, it must be detected and agreed upon by some distributed protocol. The ring then has to be reconfigured to bypass the failed process.

- *Loss of token:* When a token loss is detected (e.g. by timeout), a new one can be generated by using *leader-election* protocols. (This will be studied later in the course.)

- *Performance:*

  - *Number of messages:* In the worst case ("light load"), n messages are sent between try_i and $crit_i$. Under "heavy load", however, only a constant number of messages per request is expected.

  - *Time:* Assume worst-case bounds: c = time spent in C, d = message delay, p = process steps. The worst-case time is $\approx (c + d + O(p)) \cdot n$. This time bound is bad because it has a $d \cdot n$ term, regardless of the load, and d may be big.

9.1.3 Lamport 1978

This paper: "Time, Clocks and the Ordering of Events in a Distributed System", is a famous one and worth reading. It introduces the idea of *logical time (ltime)*: every event that occurs in a distributed system (e.g. send, receive, local steps) is assigned a distinct *logical time* that is an element of some total ordering. One way partially ordered local times at different sites can yield total ordering is by appending the site's ID as the low-order bits to the local time, thus breaking ties. Logical time behaves like real time in the following sense:

1. The order of events at each process is consistent with the order of occurrence. (Ensured by keeping a local clock at each site and incrementing it between any two successive local events.)

2. For any message, its *send* event is ordered before its *receive* event. (Ensured by attaching a *timestamp* ts, equal to the logical time of *send*, to each message. If for the clock C_r at the receiving site: $C_r \leq ts$, then increment C_r to be $> ts$ before assigning a logical time to *receive*.)

3. Any event has only finite number of predecessors. (Ensured by incrementing the local clock by some minimum value.)

We assume that the network deliveres messages between any pair of nodes in the same *ltime* order as they were sent. For example, in the space-time diagram of Figure 9.3, message a is sent before message b, and must be received before message b. Another assumption is that every message sent is eventually received. Both assumptions can be ensured by some network protocol that uses acknowledgments and puts sequence numbers on messages.

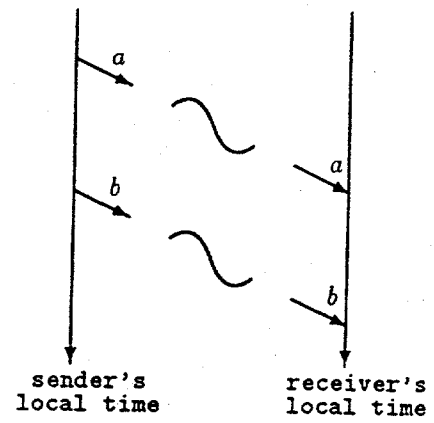


Figure 9.3: Space-Time diagram.

Being able to totally order the events can be very useful in implementing a distributed system. We shall use this method in the following algorithm to solve the mutual exclusion problem. In this algorithm, every process P_i maintains a local variable *region* as before, and for each other process P_j a local queue $queue(j)$. There are three types of messages:

- *try-msg(i):* broadcasted by P_i to announce that it is trying.
- *exit-msg(i):* broadcasted by P_i to announce that it is exiting.
- *ack(i):* sent by P_i to P_j, acknowledging the receipt of a *try-msg(j)* message.

We plan to achieve mutual exclusion by servicing requests in the *ltime* order of the broadcast event of their *try-msg*. The queues at each process behave like replicas of a global centralized queue that determines the service order. All we need now is rules for P_i telling when to send $crit_i$ and rem_i messages to $user_i$.

Rules for P_i

- $\underline{P_i \rightarrow R:}$ once an $exit_i$ occurs.
- $\underline{P_i \rightarrow C:}$ $region = T$ and the following conditions hold:
  1. Mutual exclusion is preserved.
  2. There is no other request pending with an earlier *ltime*.

  P_i can ensure that the above conditions are met by checking for each $j \neq i$:

  1. Any *try-msg* in $queue(j)$ with *ltime* < *ltime*(current *try-msg(i)*) has also a subsequent *exit-msg*.
  2. $queue(j)$ contains some message (possibly *ack*) with *ltime* > *ltime*(current *try-msg(i)*).

Properties

- Mutual Exclusion: The correctness proof is by contradiction. Assume that two processes, P_i and P_j, are in C at the same time, and (without loss of generality) that *ltime(P_i's request)* < *ltime(P_j's request)*. P_j had to check its *queue*(i) in order to enter C. The second test and our assumption on messages order preservation imply that P_j had to see P_i's *try-msg*, but by the first test it had also to see an *exit-msg* from P_i, so P_i must have already left C.

- No lockout: This property results from servicing requests in *ltime* order. Since each (request) event has finite number of predecessors, all requests will eventually get serviced.

- Complexity:

  - *Number of messages:* every request involves with sending *try-msg,ack* and *exit-msg* messages between some process and all the others, thus $3(n-1)$ messages are sent per request.
  - *Time:* for a single request in the system, with no others around, the time is $2d + O(p)$, where d is the communication delay and p is local processing. We assume that the broadcast is done as one atomic step, else, if $n-1$ messages are treated separately, the processing costs would have been linear in n.

9.1.4 Ricart & Agrawala 1981

This algorithm uses only $2(n-1)$ messages per request. It improves Lamport's algorithm (section 9.1.3) by acknowledging requests in a careful manner that eliminates the need for *exit-msg* messages. This algorithm uses two types of messages only: *try-msg* and *OK*. Process P_i sends *try-msg(i)* as in Lamport's algorithm, and can go critical after *OK* messages have been received from all the others.

Rule for sending an *OK* message

In response to a *try-msg*, a process:

- replies with an *OK* if it is not critical or trying.

- if critical: defers the reply until it exits, and then sends immediately all the deferred *OK*s.

- if trying: compares the *ltime* of its request to the one of the incoming *try-msg*. If bigger then send *OK*, else defer (i.e. allow requests with lower *ltime* only to proceed).

Properties

- Mutual Exclusion:

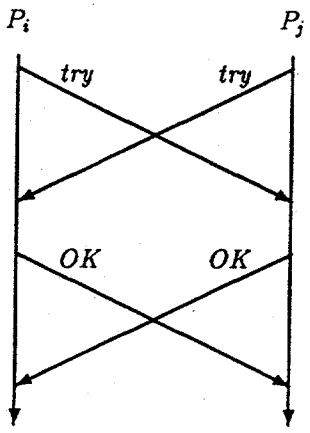


figure 9.4: both processes in C.

Using contradiction to prove correctness, assume both processes, P_i and P_j, are in C and (without loss of generality) that *ltime(P_i's request)* < *ltime(P_j's request)*. As seen in figure 9.4, P_j's *try* message has to arrive at P_i after P_i's *try*, or else our assumption on their *ltime* order would not have been correct. At the time P_i receives P_j's *try*, it is either trying or critical. In both cases, P_i's rules say it has to defer the *OK* message, thus P_j could not be in C.

- Progress: Using contradiction again, assume some execution that reached a point after which no progress is achieved. That is, at that point all the processes are either in R or T, none in C, no process changes regions any more and no message is in transit.

  Among all the processes in T after that point, assume that P_i has the request message with the lowest *ltime*. P_i is blocked forever because some other process P_j has not returned an *OK* message to it. P_j could only have deferred the *OK* because it was:

  - in C: because P_j eventually left C, it had to send the deferred *OK*.
  - in T: P_j deferred the *OK* because the *ltime* of its request was smaller than P_i's. Since P_i's request has the smallest *ltime* in T now, P_j must have completed, thus after exiting C it had to send the deferred *OK*.

9.1.5 Carvalho & Roucairol 1983

This algorithm improves on the previous one (Section 9.1.4) by giving a different interpretation to the *OK* message. When some process P_i sends an *OK* to some other process P_j, not only does it approve P_j's current request, but it also gives P_j P_i's permission to reenter C again and again until P_j sends an *OK* to P_i in response to a *try-msg* from P_i.

This algorithm performs well under light load. When a single process is requesting again and again, with no other process interested, it can go critical with no message sent! Under heavy load, however, it behaves similarly to Ricart and Agrawala's algorithm.

9.2 Exercises

1. Give careful definitions (I/O automaton schedule modules) for the following concepts, defined informally in class:

   a. A one-writer, multireader safe register with value set V, initialized at v_0.

   b. A multiwriter, multireader atomic register with value set V, initialized at v_0.

2. Prove the following facts, used in the fairness proof for Rabin's randomized mutual exclusion algorithm. Let each of n players independently draw a sequence of independent bits, with equal probability that each bit is 0 or 1, until some bit (bit k) is 0. Let $max(k)$ denote the maximum value of k among the n players. Let $b = \lceil \log n \rceil + 4$.

   a. $\text{Prob}(max(k) \leq b) \approx \frac{1}{e}^{\frac{1}{16}}$.

   b. The probability that more than one player will draw a sequence of length $max(k)$ is less than $\frac{1}{3}$.

3. Consider Rabin's randomized algorithm once again.

   a. State and prove a result of the form "with probability $f(r)$, a trying process succeeds within r rounds in which it participates".

   b. Give a similar result of the form "with probability $f(t)$, a trying process succeeds within time t". (You might base this on the probability of succeeding in a round, together with an upper bound on the time required for a round.)

4. For Lamport's distributed mutual exclusion algorithm, try to improve on the amount of local storage used, over the version of the algorithm presented in class. That is, try to condense the information that is retained, while allowing each node to exhibit the same behavior as before.

*5. Give a careful proof of the theorem stated in class, saying that arbitrary executions of shared memory systems are reducible to executions in which object invocations and responses occur consecutively.

*6. Work out a careful, complete analysis of Rabin's algorithm, using conditional probabilities.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 10: October 18

Lecturer: Nancy Lynch | *Scribe: Atul Shrivastava*

10.1 General Resource Allocation Problem

In previous lectures, the focus was on mutual exclusion problem. Mutual exclusion (which allows at most one process to use the critical section at any time) is a special case of the Resource Allocation problem. The general resource allocation problem involves n processes $p_1, \ldots, p_n$ contending for a set of resources $R_1, \ldots, R_m$.

10.1.1 Problem Description

There are two ways of describing the Resource allocation problem.

1. *Exclusion problem:* This is presented as a set of collection of processes which are not allowed to use one critical resource simultaneously. The set can be any collection of processes and should be closed under containment. We call this set the *exclusion set.*

   Example 1: Mutual exclusion can be defined as

   $\xi = \{c \subseteq \{p_1, \ldots, p_n\} :| \, c \mid \geq 2\}$

   Example 2: The k-exclusion problem (number of processes in the critical section at any time $\leq k$) can be defined as

   $\xi = \{c \subseteq \{p_1, \ldots, p_n\} :| \, c \mid \geq k+1\}$

   Example 3: For $n = 4$, let

   $\xi = \{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_4\}, \{p_3, p_4\}\}$

   Here p_1 doesn't exclude p_4, and p_2 doesn't exclude p_3. Consequently (p_1, p_4) or (p_2, p_3) can share use of the critical resource.

   The set ξ can be any arbitrary set of subsets of $p_1, \ldots, p_n$.

2. *Multiple Resource problem:* This is presented as a boolean formula (for each process p_i) describing the combination of the resources needed by p_i to enter the critical region.

   Example 4: Consider a resource allocation problem with 4 processes and 4 resources.

p_1 : $R_1 \cap R_2$
p_2 : $R_1 \cap R_3$
p_3 : $R_2 \cap R_4$
p_4 : $R_3 \cap R_4$

Here p_1 needs R_1 and R_2 to enter the critical region, etc.

Note: A resource problem can be converted into a exclusion problem. The exclusion set contains those subset of processes whose resource allocation formulae cannot be satisfied simultaneously. Thus, the exclusion set for the resource allocation problem in the preceding example is

$\xi = \{\{p_1, p_2\}, \{p_1, p_3\}, \{p_2, p_4\}, \{p_3, p_4\}\}$

The *remainder* (R), *trying* (T), *critical* (C) or *exit* (E) regions are used as before to describe the state of each process. The process behavior in each of these states is similar to that described for the mutual exclusion problem.

Progress Condition

For mutual exclusion algorithms, there is a strong notion for the progress of the system. Although its hard to state an interesting general condition for system progress, the mutual exclusion idea can be used to get some idea for system progress condition. Progress for the resource allocation problem can be described as a requirement where some process continues to make region progress, given the same notion of normal operation as for mutual exclusion.

Fairness Condition

The no lockout (or starvation) idea can be used to define fairness for the resource allocation problem.

10.2 Dining Philosophers Problem

The Dining Philosophers problem is a special type of exclusion problem, and is generally presented as a *resource allocation* problem.

10.2.1 Problem Description

There are n philosophers seated around a table. Each philosopher, is either thinking (R), hungry (T), eating (C) or just finished eating (E). In order to eat, each philosopher needs two forks; n forks are placed on the table such that there is one fork on the left and one to

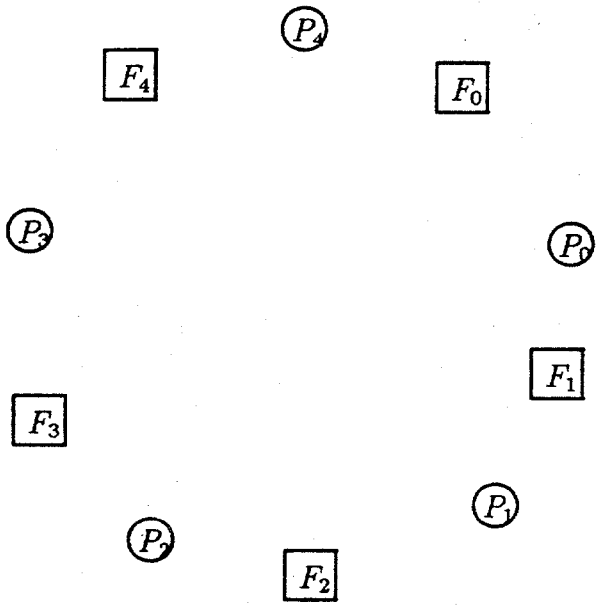


Figure 10.1: Dining Philosophers problem (n = 5)

the right of each philosopher. Each philosopher can pick up forks located immediately to his left or right only when the neighbor, with whom the fork is shared does not have the fork.

We denote each philosopher by p_i and the forks to the left of each p_i by F_{i-1} and to the right of p_i by F_i. Thus, p_i needs $F_{i-1} \cap F_i$ (p_0 needs $F_n \cap F_0$) to eat (C). After eating, each p_i puts down the both the forks (E) and resumes thinking (R). Figure 10.1 describes the seating arrangement for $n = 5$ processes.

The exclusion set for n dining philosophers is

$\xi = \{\{p_i, p_{i+1}\}, i \in \{0, \ldots, n\}\}$

Various algorithms are known that solve the Dining Philosophers problem. The first solution, presented by Dijkstra (1971) uses operating system concepts such as semaphores. Chang presented the first distributed solution to the problem. Burns' algorithm gave better time bounds for the Dining Philosopher's problem. Lynch (1981) presented a general solution to the static resource allocation problem. A randomized algorithm to solve the Dining Philosophers problem was proposed by Rabin and Lehmann (1981). All of the above algorithms use shared memory variables. Chandy and Misra proposed a solution using the message passing model. In this lecture, the first four solutions will be discussed. The other solutions will be taken up in Lecture 11.

10.2.2 Shared Memory concepts

Each process p_i uses a shared memory test and set operation to change shared variables. Each shared variable is used as a binary semaphore. Thus two operations can be performed on any shared variable s. These operations can be written as

$$P(s) : \textit{waitfor } s = 1;\ s \leftarrow 0$$
$$V(s) : \textit{waitfor } s = 0;\ s \leftarrow 1$$

Although each operation locks the variable before performing the indivisible conditional test and set, the variable is unlocked immediately, irrespective of the test condition being found to be true or false. These concepts came out of operating systems where the scheduler checks for the condition and if not found to be true, puts the process in a sleep state in some queue. When the value of the shared variable changes, the scheduler checks to see the processes whose conditions are now satisfied, and then wakes those processes.

10.3 A simple approach that deadlocks

The algorithm is conceptually simple, each process grabs the left fork first and then picks up the right fork. After getting both the forks, it $\rightarrow$ C. When a process leaves C, it puts down both forks before entering R.

10.3.1 Properties of the algorithm

We check the mutual exclusion and deadlock freedom properties for the above algorithm.

Mutual Exclusion

To $\rightarrow$ C, a process p_i has to pick up both its left and right forks. The two P-operations guarantee that the values of $FORK_{i-1}$ and $FORK_i$ are both 0 when $p_i \rightarrow$ C. Thus when p_{i-1} (p_{i+1}) tries to grab $FORK_{i-1}$ ($FORK_i$), it will be blocked when it executes the second (first) P-operation in its code. Thus mutual exclusion is preserved.

Deadlock Possible

This algorithm however does not guarantee deadlock freedom. Consider the sequence of events starting with each process in R. Now, each process wakes up and grabs its left fork at the same time. At this point each process tries to get its right fork but since all forks are already picked up, all process starve forever and the system cannot make any progress (i.e. no process can change regions).

Shared variables:

$\forall i, FORK(i) \in \{0,1\}$, written by and read by several processes, initially 1

Code for p_i:

P($FORK_{i-1}$)
P($FORK_i$)
unlock;

** Critical Region **

lock;
V($FORK_{i-1}$)
V($FORK_i$)
unlock;

** Remainder Region **

lock;

Figure 10.2: A simple solution

10.4 Dijkstra's Solution

The shared variables used in the solution are *not* associated with forks. Binary semaphore *mutex* is shared by all process and is initially 1; *control* is a multi-reader multi-writer array, where $\forall i$, $control(i)$ initially 0 and is read and written read by p_{i-1}, p_i and p_{i+1}; *sem* is an array of binary semaphores, initialized to 0.

$$control(i) = \begin{cases} 0 & \Rightarrow p_i \in R \\ 1 & \Rightarrow p_i \in T \text{ but unable} \rightarrow C \\ 2 & \Rightarrow p_i \text{ allowed} \rightarrow C \text{ or } p_i \in C \end{cases}$$

mutex is used so that a group of operations can be done indivisibly. $sem(i) = 1$ tells p_i that it can $\rightarrow C$.

If any p_i (with $control(i) = 1$), finds that $control(i-1) \neq 2 \cap control(i+1) \neq 2$ (indivisibly), then $p_i \rightarrow C$ and $p_i$sets$control(i) \leftarrow 2$. The procedure **TEST**(i) checks for this condition. Its description is included in Figure 10.3.

To incorporate indivisibility during executing TEST(i), the call to procedure is always preceded by a $P(mutex)$ operation, which when "successfully" completed guarantees that no other process can interfere while p_i is in TEST(i). A $V(mutex)$ operation after TEST(i) later allows other processes to access the shared variables.

In the code for p_i presented in Figure 10.3, there are no lock and unlock statements guarding modifications to the shared variables. But the use of shared variable *mutex* in the code guarantees the following:

1. No process can access a shared variable while p_i is executing TEST(i).
2. No process can access $control(i)$, whenever p_i is modifying $control(i)$.

10.4.1 Properties of the algorithm

Mutual Exclusion

To $\rightarrow C$, p_i has $control(i) = 2$. This condition is only true when $control(i-1) \neq 2 \cap control(i+1) \neq 2$. Since $\forall i$, testing and setting neighbors' $control(i) \leftarrow 2$ is performed indivisibly, mutual exclusion is preserved.

Progress

The only impediments to the progress of any process are the P and V operations. Hence, it is enough to show that if all processes cannot get stuck at a P- or V-operation forever. We establish this proof in two parts. First, we show that processes can't get stuck during a

Shared variables:

$mutex$ is a binary semaphore, initially 1
$\forall i, control(i) \in \{0, 1, 2\}$, written by and read by several processes
$\forall i, sem(i)$ is a binary semaphore, initially 0

Procedure TEST(i):

if $control(i-1) \neq 2$ **and** $control(i+1) \neq 2$ **and** $control(i) = 1$ **then**
  $control(i) \leftarrow 2$
  V($sem(i)$)

Code for p_i:

P($mutex$)
$control(i) \leftarrow 1$
TEST(i)
V($mutex$)
P($sem(i)$)

** Critical Region **

P($mutex$)
$control(i) \leftarrow 0$
TEST($i-1$)
TEST($i+1$)
V($mutex$)

** Remainder Region **

Figure 10.3: Dijkstra's Dining Philosophers Algorithm

V-operation, and then show similar property for the P-operation. Note that P and V are parity operations on *mutex*.
Claim: No process can get stuck during a V-operation.

1. A $V(mutex)$ can only be executed by one process, namely the one (say p_i) that has most recently successfully completed $P(mutex)$. Thus p_i will successfully execute $V(mutex)$, since no other process could have changed the value of *mutex* to 1.

2. $V(sem(i))$ is executed in **TEST**(i), only if $control(i) = 1$, when tested. To prove a process can't get stuck at $V(sem(i))$, it is sufficient to show that $control(i) = 1 \Rightarrow sem(i) = 0$: A process p_i sets $control(i) \leftarrow 1$, when it enters T. Note that $sem(i)$ is initially 0, and $V(sem(i))$ is the only operation that changes $sem(i) \leftarrow 1$. Whenever $V(sem(i))$ is executed in **TEST**(i), $control(i) \leftarrow 2$ just before it. Furthermore, $sem(i)$ is reset to 0 before $p_i \rightarrow C$ and $sem(i)$ remains unchanged until after $p_i \rightarrow T$, once again.

Claim: No process can get stuck during a P-operation.

1. If all processes get stuck at $P(mutex)$, parity access on $mutex \Rightarrow mutex = 1$. Thus, one of the process will successfully complete $P(mutex)$ operation.

2. *Claim:* If p_i get stuck at $P(sem(i)) \Rightarrow control(i) = 1$.

   If $control(i) = 2$, it must have happened that $control(i) \leftarrow 2$ and $V(sem(i))$ were executed successfully and indivisibly, setting $sem(i) = 1$. Thus, there is no way that $sem(i)$ can be reset to 0 and hence $P(sem(i))$ is successfully completed. $P(sem(i))$ is successfully executed when $control(i) = 2$. So, p_i can only be stuck at $P(sem(i))$, when $control(i) = 1$.

Claim: If p_i is at $P(sem(i))$ with $control(i) = 1$, then

$$\text{Either } p_{i-1}(p_{i+1}) \text{ is in } \begin{cases} C \cup E \\ \text{or} \\ T \text{ with its } control = 2 \end{cases}$$

Proof: When $p_i \rightarrow T$, $control(i) = 1$. The only way p_i could be stuck at $P(sem(i))$ is for **TEST**(i) to have failed. Otherwise, $V(sem(i))$ would have been executed. Therefore,

$$\Rightarrow (control(i-1) = 2 \vee control(i+1) = 2)$$

If the neighbor with $control = 2$ is still in the system, then the result holds. Otherwise, when it left the system (in E), it executed **TEST**(i). If this **TEST**(i) failed, the other neighbor must have $control = 2$ at that time. This argument can be continued *ad infinitum.*

Fairness

Consider when p_i gets stuck at $P(sem(i))$ (with $control(i) = 1$). This can happen if either $control(i-1)$ or $control(i+1) = 2$. Thus p_{i-1} (or p_{i+1}) can $\rightarrow C$. When p_{i-1} (or p_{i+1}) leaves C, it executes **TEST**(i), which fails if $control(i-1)$ (or $control(i+1)) = 2$. If this continues *ad infinitum* i.e. p_{i-1} and p_{i+1} time their entries such that TEST(i) always fails, p_i will starve forever. Therefore, the solution is not fair.

Formulating a stronger correctness condition for a general resource allocation problem is posed as an exercise at the end of Lecture 11. A stronger notion of 'utilization' for any algorithm that solves the dining philosophers problem is needed to capture the concurrency that is inherent in the problem. For example, if $p_i \in T$; and if $(p_{i-1} \vee p_{i+1} \in R)$; then p_i eventually $\rightarrow C$, assuming normal operation of p_i. (All other processes can stop anywhere except in the indivisible section of their code i.e. after $P(mutex)$ and before $V(mutex)$).

10.5 Chang's Algorithm

Dijkstra's solution although correct, is centralized and does not have the fairness property desired. Thus, there is need to look at fair, totally distributed algorithms for the dining philosophers problem and resource allocation algorithms in general.

The next two algorithms solve the dining philosophers problem only. These algorithms will include fairness in addition to mutual exclusion and no deadlock. In these solutions, the shared variables ($FORK_i$) used by each process are associated with the resources (FORKS).

Each process picks up a fork in a non-deterministic fashion (essentially using the simple algorithm presented in Figure 10.2, which may result in a deadlock. To break the deadlock, some processes need to relinquish resources that they hold. But if symmetry is not broken, then all processes may put down their forks at once, only to start all over again. So, this solution still results in deadlock. Thus something additional is needed to detect deadlock and break the symmetry, in order to achieve progress.

Idea: When deadlock is detected, use Use Le Lann's token passing algorithm. The process with the token releases all resources it holds. Progress is insured and no lockout is guaranteed. Additional Overhead: Each shared variable has a queue which holds all processes requesting the fork (for dining philosophers case maximum queue length ≤ 2). Also, arbitrary message passing between neighbors needs to be allowed in order to first determine that deadlock has occurred and later take actions to break the deadlock. Thus, each process has additional code (actions for deadlock detection and continuation after a deadlock). Each process p_i on relinquishing its resource (while still in T), puts itself on the back of the queue for the resource it relinquished (p_i will keep itself in the queue until it $\rightarrow C$).
Example 5: Each process holds its left fork and waits for the right fork held by its neighbor

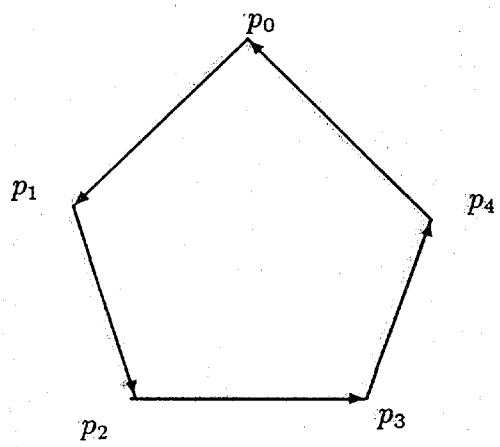


Figure 10.4: Deadlock condition

on the right as shown in Figure 10.4. Now assume that p_4 relinquishes $FORK_3$ and places itself in the queue of processes waiting to use $FORK_3$ and $FORK_4$, as shown in Figure 10.5. Since p_3 has all resources it needs, it can now $\rightarrow C$, eat and eventually release its resources. This results in a 'wave' of consumption that eventually allows each process, including p_4 to $\rightarrow C$ (eat).

10.5.1 Properties of Chang's algorithm

The algorithm can be viewed as a composition that works in two phases. In the first phase, all processes execute the code for the simple algorithm until the system deadlocks. On deadlock detection, all processes execute the token passing code as per Le Lann's algorithm.

Mutual Exclusion

Since mutual exclusion is preserved by Dijkstra's algorithm, mutual exclusion is preserved by Chang's algorithm. This is because the only difference is that processes occasionally give up their resources.

Progress

Progress in the original algorithm occurs is guaranteed until the deadlock occurs. On deadlock all processes revert to token passing in a unidirectional ring. The token passing algorithm results in some process relinquishing all its forks. Thus, the waiting chain is broken and the

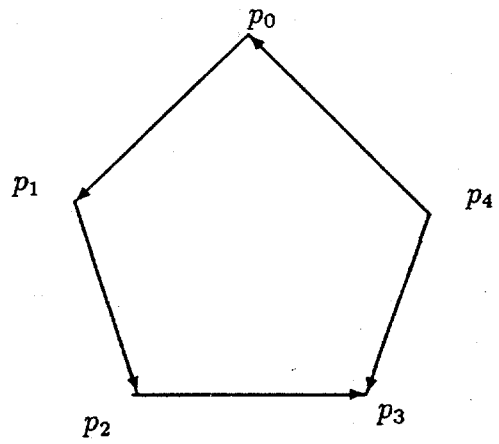


Figure 10.5: Breaking the deadlock

process whose resource requirements are satisfied can $\rightarrow C$. In fact a stronger condition of bounded waiting times in T can be calculated. Thus progress for the composite algorithm is guaranteed.

Lockout

In phase one, a process p_i can be blocked on executing any of the two P-operations before it proceeds to eat if one of its neighbor p_{i-1} or $p_{i+1} \in C$. When the neighbor leaves C, p_i is woken up and p_i completes the P-operation that caused it to be blocked. Thus if deadlock condition doesn't occur, no process can starve forever. The deadlock condition results in the start of token passing. After token passing begins, the waiting time for each process before it gets to $\rightarrow C$ (eat) is bounded by $n * c +$ *token passing time*. Thus the composite algorithm is fair.

Shortcomings

All processes are waiting in the deadlock condition. Token passing breaks the deadlock but each process $\rightarrow C$ sequentially. Thus the concurrency inherent in the problem is not utilized. This results in a worst-case waiting time of $O(n)$.

10.6 Symmetry Impossibility Result

Theorem 10.1 *There exists no symmetric distributed solution to the Dining Philosophers problem that guarantees the mutual exclusion, progress and no lockout properties.*

Proof: Assume there exists a symmetric, distributed algorithm with mutual exclusion, progress and no lockout properties. Assume a schedule S in which the following is the order of execution of processes (round robin scheduling). Assume all processes and shared variables to be in the same state initially.

$$\underbrace{1, 2, \ldots, n}_{1 \text{ round}}, 1, 2, \ldots, n, \ldots$$

A round then consists of n steps, the first step is taken by p_1, the second by p_2 and n-th and last step of the round taken by p_n. In the first step p_1 accesses its left variable and changes its value from v_1 to v_2. Then, $p_2, \ldots, p_n$ follow p_1 (since the algorithm step is symmetric). Therefore, after first round, for all i, p_i's are in the same state and all shared variables have the same value.

Inductively, this happens for each round, since the algorithm is symmetric. Thus, if when $p_1 \rightarrow C$, $p_2, \ldots, p_n$ follow and $\rightarrow C$ in the same round. This results in the mutual exclusion property being violated, and a contradiction. ■

10.7 Improving Time Bounds for Dining Philosophers

Chang's solution is not optimal in terms of time taken (worst case) by a process when it $\rightarrow T$ (gets hungry) until it $\rightarrow C$. As mentioned, the concurrency is lost as soon as processes start the token passing phase after a deadlock. The next two algorithms take steps in improving the time bounds.

10.8 Burns' Approach

Although, there may be a large number of nodes (n) and resources (r) in a system, each process uses a small number of resources and each resource is used by a small number of processes. Each process' need for resource is also known a priori. Thus intuitively, the worst case time bound for each process (waiting time for resources) should be independent of n.

Burns' algorithm is a step in this direction. The solution is asymmetric in that processes behave differently when taking steps (accessing a shared variable) in their code. The solution for n being even is presented in Figure 10.6. For n odd, refer to Exercise 2 at the end of Lecture 11.

Idea: Let even numbered processes pick their left forks first and odd numbered processes pick their right forks first. Like Chang's solution, shared variables correspond to resources

($FORKS$). Each $FORK_i$ has a flag variable indicating whether it is available for use. It also has a queue to allow processes waiting for the resource to get in line.

10.8.1 Properties of Burns' Algorithm

Mutual Exclusion

Each shared variable is a binary semaphore and the variable is incremented (decremented) using V (P) operation indivisibly. Thus, p_i needs $FORK_{i-1} = 0$ and $FORK_{i+1} = 0$ to $\rightarrow C$. If p_{i-1} (p_{i+1}) $\in C$, then $FORK_{i-1}$ ($FORK_i$) $= 0$ so p_i has to wait until p_{i-1} (p_{i+1}) does a V-operation on $FORK_{i-1}$ ($FORK_i$) and wakes up p_i. This can only happen when p_{i-1} (p_{i+1}) leaves C. Thus, mutual exclusion is satisfied.

Fairness (which implies progress)

We will bound the time that a process may have to wait $\rightarrow C$ from the time it $\rightarrow T$.

Let $T(n)$ = Time taken by $p_i \rightarrow C$ since it $\rightarrow T$ (worst-case)

$S(n)$ = Time taken by $p_i \rightarrow C$, since p_i first tries to procure its second fork (worst-case)

Assuming that single-step time $\leq s$; time in critical region $\leq c$.

Case 1: p_i immediately gets its first fork (after one step) and then its time $\rightarrow C$ is just the time taken to procure second fork. This gives us,

$$T(n) \leq s + S(n)$$

Case 2:

$T(n)$	$\leq$	s	neighbor gets the fork p_i wants (also neighbors first fork)
		$+S(n)$	neighbor getting his second fork
		$+c$	time taken for neighbor to eat
		$+2s$	time taken by neighbor to put forks down
		$+s$	time taken by p_i to pick first fork
		$+S(n)$	p_i's time to get its second fork

$$T(n) \leq 4s + 2S(n) + c \tag{10.1}$$

To calculate $S(n)$:

$S(n)$	$\leq$	c	neighbor beats you to the second fork also neighbors second fork, by odd-even rule
		$+s$	time taken by p_i to pick up fork

Substituting for $S(n)$ in Equation 10.1, we get the worst-case waiting time that is independent of n:

Shared variables:

$\forall i, control(i) \in \{0,1,2\}$, written by and read by several processes
$\forall i, FORK_i$ is a binary semaphore, initially 0. $FORK_i$ is written and read by p_{i-1} and p_i

Code for p_{2i}:

```
P(FORK_{2i-1})          (* wait for left fork *)
P(FORK_{2i})            (* wait for right fork *)

 ** Critical Region **

V(FORK_{2i-1})          (* release forks *)
V(FORK_{2i})            (* in any order *)

 ** Remainder Region **
```

Code for p_{2i+1}:

```
P(FORK_{2i+1})          (* wait for right fork *)
P(FORK_{2i})            (* wait for left fork *)

 ** Critical Region **

V(FORK_{2i+1})          (* release forks *)
V(FORK_{2i})            (* in any order *)

 ** Remainder Region **
```

Figure 10.6: Burns' Dining Philosophers Algorithm (n even)

$$T(n) \leq 6s + 3c \tag{10.2}$$

10.9 Lynch's Solution

Lynch presents a solution for all conjunctive resource allocation problems. The algorithm thus does not solve k-exclusion where multiple alternatives are allowed. Each process' needs for resources are known *a priori.*

Each resource uses a flag variable, shared by all processes that use the resource. A FIFO queue is also associated with each resource in which each process enters its request for the resource in order. Each process can wait (queue up) for maximum of one resource at a time.

Definition Let p_i be currently waiting for resource r that is currently used by p_j. Then, p_i is in *waiting chain* .

Idea: The algorithm may be designed to allocated resources in a fashion such that length of the longest waiting chain is small. This will result in smaller waiting times needed by a process for each resource needed to $\rightarrow C$, thus improving the bound on waiting times.

Basic Strategy: Using the "Hierarchical resource allocation" strategy, assign to the resources in the system a global ordering. Each process p_i seeks the resource with the highest ID in the ordering, which it needs and and doesn't already have (Note that this is a generalized form of L-R strategy).

Theorem 10.2 *Consider a system with totally ordered resources* $(r_1, \ldots, r_m)$. *Processes* $(p_1, \ldots, p_n)$ *seek a conjunctive set of resources to* $\rightarrow C$. *Each resource has a queue in which processes place their request (in order of their request times). When a resource becomes available, the process at the head of the queue removes itself from the queue and assumes control of the resource. Each process can seek only one resource at a time (until all it needs are satisfied). Each process can queue up only for resources that it needs and doesn't already have. Also, the resource sought by a process at any time has the highest ID (from the resource ordering). In such a system, assuming that processes release all resources when they depart from* C, *the system can never deadlock. Moreover, each process will eventually get all the resources it needs.*

Proof: Assume such a deadlock occurs (no process can $\rightarrow C$). Since each process can appear in at most one resource queue, the process (say p_i) at the head of the highest priority queue (say r_j) can never wait for any other resources. So, p_i grabs r_j. Then, all processes behind p_i "move up" in priority. Eventually p_i leaves C and a new highest priority process can proceed. ■

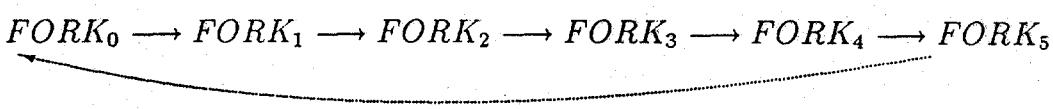


Figure 10.7: Resource Allocation

Now consider five dining philosophers (DP(5)) with total ordering of resources. Let $FORK_0$, $FORK_1$, $FORK_2$, $FORK_3$ and $FORK_4$ be the ordering of the resources in descending order.

Worst Case execution:

p_4 arrives and picks $FORK_3$ and $FORK_4$
p_3 arrives and picks $FORK_2$, waits for $FORK_3$
p_2 arrives and picks $FORK_1$, waits for $FORK_2$
p_1 arrives and picks $FORK_0$, waits for $FORK_1$
p_0 arrives and waits for $FORK_4$

Waiting Chain

$$p_0 \longrightarrow p_1 \longrightarrow p_2 \longrightarrow p_3 \longrightarrow p_4$$

This results in long waiting chains $O(n)$ and consequently long delays. Consequently a better strategy for resource allocation is needed.

10.9.1 Resource Allocation Strategy

Looking at the resource allocation problem for DP(6), results in the graph shown in Figure 10.7.

The vertices $FORK_i$ and $FORK_{i+1}$ in the resource allocation graph are connected by edge p_{i+1}, since process p_{i+1} needs both $FORK_i$ and $FORK_{i+1}$ to $\rightarrow C$.

Color the above graph such that no two adjacent vertices have the same color using the minimum number of colors. In general, the minimization problem is NP-Complete (but 'small' number of colors will do). Now, totally order the colors. The resources now

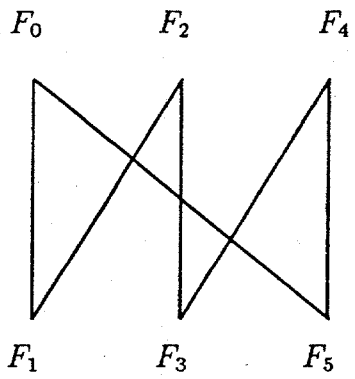


Figure 10.8: Resource Allocation for DP(6)

form a partial order where $FORK_i < FORK_j$ iff $FORK_i$ and $FORK_j$ are adjacent and $color(FORK_i) < color(FORK_j)$. Thus DP(6) results in a resource allocation graph, shown in Figure 10.8.

Thus, $FORK_0, FORK_2, FORK_4$ have color 0, and $FORK_1, FORK_3, FORK_5$ have color 1. This results in a partial order. For the dining philosophers problem, the two $FORKS$ needed by a process are colored differently and are therefore ordered. Each process now picks up resources it needs in increasing order according to the partial ordering, i.e. increasing order of colors (same as any total ordering).

For the dining philosophers problem, the partial ordering of resources results is identical to the L-R alternate strategy (Burns' algorithm). - hence this is a special case of the general hierarchical resource allocation strategy.

Algorithm Properties

The partial ordering strategy for dining philosophers problem is deadlock free and is fair to all processes, because its an instance of the Hierarchical Resource allocation strategy. This result has already been shown for Burns' algorithm.

Let l denote the length of the biggest waiting chain. Let c be the number of colors in a resource allocation graph. Then, in the worst case,

$$l = n$$

Proof: Process p_i is at the end of the waiting chain, waiting for resource with lowest color number. This resource is held by process by p_j. p_j in turn could be waiting for resource with the next higher color etc. This will continue till some process p_r waiting for the resource

with highest color number. There cannot be more processes waiting for resources in this waiting chain. ∎

For dining philosophers problem (n even), $l = 2$.

10.9.2 Bound on Waiting times

The worst-case waiting time for a colored graph, with

s = upper bound on step time
c = upper bound on time spent in its critical section
k = total number of colors
m = maximum number of users for a resource

can be bounded by

$$m^k c + k m^k s$$

Proof: Let $T_{i,j}$ be the time taken by a process to $\rightarrow C$ from the time it begins to occupy position j, $(0 \leq j \leq m-1)$ of a queue for color i, $(0 \leq i \leq k-1)$.

The equations for various $T_{i,j}$ can be written down as follows:

$T_{k-1,0}$	=	0	process reaches the front of the highest color queue
$T_{i,j}$	=	c	time for process in C to leave
		$+ks$	the process entering C to pick its resources
		$+T_{i,j-1}$	time taken by process in front to reach C
		$+T_{i,0}$	time taken by process to $\rightarrow C$ after getting resource with color i $(j > 0)$
$T_{i,0}$	=	s	process grabs resource i
		$+T_{i+1,m-1}$	time taken to reach C starting from the back of the queue for color $i+1$

In the worst case all processes need k resources and all resources are used by m processes.

The total time taken by a process since it $\Rightarrow T = T_{0,m-1}$. Plugging in the equations in the following sequence.

$$\begin{aligned} T_{0,m-1} &= (c + ks)m + m\,T_{0,0} \\ T_{0,0} &= s + T_{1,m-1} \end{aligned}$$

$$\begin{aligned} \ddots \quad & \quad \ddots \\ T_{k-1,m-1} &= (c+ks)m + m\,T_{k-1,0} \\ T_{k-1,0} &= 0 \end{aligned}$$

Assuming $s << (c+ks)m$ (small step time), and $(m >> 1)$, we get

$$\begin{aligned} T_{0,m-1} &\leq (c+ks)\sum_{a=0}^{k-1} m^a \\ &\leq m^k c + k m^k s \end{aligned}$$

■

Again, the worst-case bound is independent of n. Notice that the bound is not tight for the dining philosophers problem. However, for the general resource allocation problem, exponential behavior for waiting times is possible.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 11: October 20

Lecturer: Nancy A. Lynch | *Scribe: Rick Stille*

We continue our discussion of dining philosophers algorithms.

11.1 Rabin-Lehmann dining philosophers

The Rabin-Lehmann algorithm (Figure 11.1) is a randomized solution to the dining philosophers problem which guarantees both mutual exclusion and progress. A modification of this algorithm (not presented here) also guarantees no lockout. In this algorithm, all processes are identical. Randomization is used to break the symmetry.

```
Code for p_i where opp (s) = {R if s = L, L if s = R};
 do forever
    draw random s from {R,L} with equal probability;
    wait for s to be free;
    pick s up;
    if opp(s) is free, then
       pick s up;
       exit;
    else
       put s down;
    end if;
          * * critical region (philosopher is eating) * *
    put down both forks;
       * * remainder region (philosopher is thinking) * *
```

Figure 11.1: Rabin-Lehmann algorithm

Individual fork accesses are indivisible. A process chooses randomly each time through the loop, but only waits for the first fork, and checks only once for the second. Then, if unsuccessful, it starts over with a new random choice.

11.1.1 Mutual Exclusion

Mutual exclusion is obvious. It is not a probabilistic statement.

11.1.2 Progress (with Probability 1)

Let adversary A be a function mapping finite executions to process identification numbers, identifying the process which is to take the next step. Note that this adversary is more powerful than those we have seen in previous lectures. For example, our earlier adversaries were not allowed to see the results of previous random draws. We call A *normal* if it only generates normal executions, i.e., executions which give fair turns to all processes. We will only consider normal adversaries.

Definition Let *exec(A,D)* be the execution generated by adversary A and sequence D of random draws. Our assumption says that this execution is normal, for all D.

Probabilities

Assume that each draw gives R or L with equal probabilities of $1/2$. This gives a probability distribution for all infinite draw sequences D. For each A, the distribution on the set of draw sequences yields a corresponding probability distribution on the set of executions generated by A (i.e., a D distribution yields an $exec(A, D)$ distribution).

Our goal is to show that the probability of deadlock is zero. It suffices to show that for any finite execution e, the probability that someone changes region after e (if anyone is not in R just after e) is equal to one. This is a conditional probability with respect to prefix $= e$. We will prove this by the following series of lemmas.

Definition Let *dead* be the set of executions with prefix e in which no one proceeds to C, but someone is in T.

We wish to show that $\Pr(dead) = 0$.

Remark 11.1 *In any execution in dead, there are infinitely many fork pickups.*

Proof: Suppose not. Then consider what happens after the last fork pickup. All processes are stuck in T, so all forks ever picked up eventually get put down. At this point there is nothing to stop a process from picking up. ∎

Remark 11.2 *If p chooses infinitely often, then with probability 1 it chooses L infinitely often and R infinitely often.*

Proof: Suppose, without loss of generality, that p chooses L only finitely often in some execution f. Then there must exist a finite prefix e of f after which p chooses only R and does so infinitely often. But the probability that p will choose R in any draw is $1/2$ and thus the probability that p will choose R n times consecutively is $(1/2)^n$. It follows that the probability P of doing so an infinite number of times is $\lim_{n\to\infty}(1/2)^n = 0$ so the probability of choosing both R and L infinitely times is $1 - P = 1 - 0 = 1$. ∎

Lemma 11.3 *Let p be q's neighbor. If p picks up a fork infinitely often and q does not, then with probability 1, p eats (goes to C) an infinite number of times.*

Proof: Assume q is p's right neighbor, without loss of generality. If p picks up a fork infinitely often, it chooses infinitely often. Then with probability 1, p chooses L infinitely often. Each time p does this, it must succeed in getting the fork (or else it would be stuck). Then it looks for its right fork. In each of these executions, q eventually stops picking up forks, so p will thereafter always succeed in getting the right fork and will eat. ■

Lemma 11.4 *In any execution in dead, every process picks up a fork an infinite number of times with probability 1.*

Proof: By Remark 1, there are infinitely many fork pickups. The only way that this can fail is if there is a positive probability set of executions in *dead* in which some process p picks up a fork infinitely often and neighbor q picks up a fork only finitely often. But Lemma 11.4 implies that this is impossible (since p could eat and these executions are deadlocked). ■

So with probability 1, deadlocking involves all processes continuing to pick up forks infinitely often and making infinitely many random left choices and infinitely many random right choices.

Definition We define a *good* configuration (a point in a particular execution) to be one in which there are two processes p and q in T, where p is q's left neighbor, p's latest random draw is L, and q's latest random draw is R.

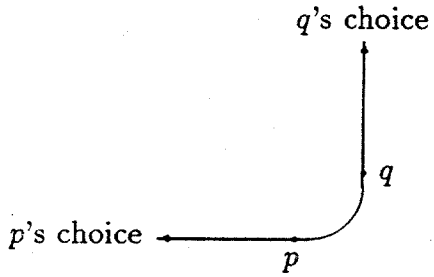


Figure 11.2: A *good* configuration. Arrows indicate the latest random draw by a process.

Lemma 11.5 *Starting from a good configuration, if every process picks up forks infinitely often, then the probability that at least one of p or q eats no later than two random draws later is greater than or equal to $1/2$.*

Proof: We use a case analysis, based on whether each is before or after the point (call it point m) of examining the second fork.

(a) *Both are before m.*
Then both wait and one is guaranteed success. Whichever gets the shared fork first succeeds. Thus, in this case the probability of success is 1.

(b) *Both are after m.*
If either has succeeded in getting the second fork, then we are done. If neither has succeeded, then each of them must have been holding the shared fork sf when the other checked it. We will show that this is impossible. Suppose, without loss of generality, that p examined sf first, finding that q had it. Then, while p stayed in T, q went back and chose R again, then examined L. But q would have succeeded in picking up sf in this case, contradicting our assumption that neither has succeeded.

(c) *p is before m and q is after m.*
If q has succeeded, we are done. If not, q chooses again and with a probability of $1/2$ chooses R again. At this point p and q will both be before m and case (a) ensures success.

(d) *q is before m and p is after m.*
If p has succeeded, we are done. If not, p chooses again and with a probability of $1/2$ chooses L again. At this point p and q will both be before m and case (a) ensures success. ■

Definition Two prefixes e and f (where f is a finite extension of e) are *disjoint* if all processes draw in the sequence which, when appended to e, produces f.

Lemma 11.6 *If every process picks up forks infinitely often, then with probability 1, there are infinitely many disjoint prefixes of this execution, each of which results in a good configuration.*

Proof: Suppose to the contrary that there are only finitely many disjoint prefixes. Then there must exist some point in the execution after which there are no more extensions which produce a disjoint prefix. But this implies that there is at least one process that does not pick up any more forks after this point in the execution, contradicting our assumption that every process picks up forks infinitely often. Thus there must exist infinitely many disjoint prefixes.

Since, by Remark 11.1, each process chooses L infinitely often and R infinitely often, with probability 1 infinitely many of these disjoint prefixes result in a good configuration. ■

We now use the above lemmas to prove the following theorem.

Theorem 11.7 *Prob(dead) = 0.*

Proof: Suppose to contrary that Prob(*dead*) > 0. Then Lemma 11.4 implies that, with probability 1 , relative to *dead*, every process picks up a fork infinitely often. Lemma 11.6 then implies that with probability 1, there are infinitely many disjoint prefixes resulting in good configurations. Lemma 11.5 then implies that, with probability 1, some process eventually proceeds to C. But Prob(*someone eventually eats*|*dead*) = 0 by definition. Therefore, Prob(*dead*) = 0. ■

Fairness

This algorithm does not guarantee fairness but a later modification does.

11.2 Chandy-Misra dining philosophers

The Chandy-Misra algorithm is another solution to the general *conjunctive resource allocation* problem. It guarantees *mutual exclusion, progress,* and *no lockout* as in the Lynch solution. The time performance can be worse in the worst case, but they have developed a somewhat more dynamic version, called Drinking Philosophers (see Section 11.3 where processes can require different resources each time they enter T.

The present algorithm uses a *message-passing model* where each process has a *fixed set of resource requirements.*

We will make two restrictions:

1. Any pair of processes shares at most one resource. (This restriction is easy to remove.)
2. Each resource shared by at most two processes. (It would take some work to remove this restriction. One would need to modify the algorithm.)

Note the close similarity to the Ricart & Agrawala and Carvallho & Roucairol algorithms.

11.2.1 Description of the algorithm

A fork shared by two processes is always either *clean* or *dirty*, initially all are dirty. While being used to eat, it is dirty and remains dirty until cleaned. A philosopher can clean a fork only when mailing it and can only mail a clean fork.

While in R

All forks held are dirty.
Satisfy all requests received (i.e., clean and send forks requested).

While in T

Forks received since entering T *are clean, all others are dirty.*
Request every fork needed and do not have (including any previously released). If a request arrives, satisfy it if it is for a dirty fork. Otherwise, defer the request. When all forks needed are held, make all forks dirty and proceed to C

While in C

All forks needed are held and are all dirty.
Defer all requests received.

While in E

All forks held are dirty.
Satisfy all deferred requests and all requests that arrive while in E. Proceed to R when all requests have been satisfied.

11.2.2 Correctness

Proving correctness depends on preserving a nice invariant property of a certain dynamically changing digraph (directed graph) H. If G is the graph with

- processes at the nodes, and
- edges between processes that share a resource (forks associated with edges),

then we get H by directing each edge of G as follows. Direct edge (p,q) exactly if the fork associated with the edge is

1. at p and dirty, or
2. in transit from p to q, or
3. at q and clean.

The notation (p,q) means that q has priority over p for the resource.

Mutual Exclusion

The invariant to preserve is that *H is acyclic.* So we start with an initial condition that satisfies this, i.e., breaks the symmetry of the system. We argue as follow that the algorithm preserves acyclicity:

The only change occurs when a process dirties a clean fork, i.e., when it eats. In this case it must have all forks and dirties them all at once. So all edges incident on that process get directed away from it. Therefore it cannot belong to a cycle.

Progress

Definition The *height* of a process p in an acyclic graph H is the maximum length of the directed path in H leading away from p.

Claim 11.8 *The Chandy-Misra dining philosophers algorithm guarantees no lockout.*

Proof: We will show inductively on k that in any H for a reachable configuration, any process in T in that configuration and with height k eventually proceeds to C.

Height $k = 0$.
Then all edges are incoming, so eventually p will get all (clean) forks and proceed to C.

Inductive step, height $= k$, for p in T
For all incoming edges, p will eventually get clean forks (and will keep them until it proceeds to C. For each outgoing edge (p, q), it must be that q is of height $\leq k - 1$. For each such edge, the associated fork must be in one of the following catagories, by definition:

(a) Fork is at p and dirty.
(b) Fork is in transit from p to q.
(c) Fork is at q and is clean.

For case (a), if the fork does not leave before p proceeds to C, then p cannot be blocked by this fork. If the fork does leave, then we are in case (b) or case (c).

For cases (b) and (c), q is in T at some time before p proceeds to C. But in this case q has height $\leq k - 1$ and eventually proceeds to C by inductive hypothesis. Eventually, q proceeds to E, which causes the edge to get redirected toward p. Then p eventually gets the fork and keeps it. ■

11.3 Chandy-Misra drinking philosophers

This is an extension by Chandy and Misra of the dining philosophers problem to a more dynamic problem in which processes do not necessarily require their entire possible set of resources each time, but rather some arbitrary subset.

The first idea for solving such a problem might be to modify the previous solution very slightly - where each process just requests and waits for those resources it wants, but the following example shows that this does not work.

11.3.1 Example

Suppose we have five philosophers in a ring, each with two forks (one on each side) as possible resources. Begin with an acyclic H as shown in Figure 11.3, where a dirty fork is located at the tail of each arrow.

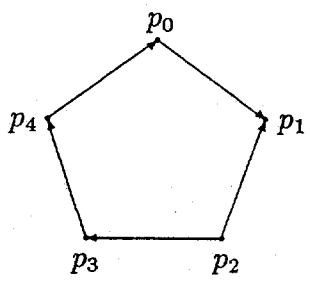


Figure 11.3: Acyclic graph H. All forks are dirty.

First, all philosophers enter wanting only their left forks. They all can get them: surely those with in-edges get them; but also p_1, since p_1 has its fork dirty and p_0 does not want it. So they all can use their left forks at once, dirtying them in the process. But this dirtiness orients the edges to the left, creating a cycle!

Thus the invariant breaks down and the next time they all might come in wanting both forks, creating the possibility of deadlock.

11.3.2 Lynch version of Chandy-Misra solution

The Chandy-Misra solution can be expressed in a cleanly separated way, though this is not how they present it. We will use their dining philosophers algorithm as a subroutine to insure that every reachable state of the system preserves the acyclicity of the graph for this subroutine (though not for the main algorithm). The architecture of our proposed solution is shown in Figure 11.4.

The dining philosophers algorithm executes using its own messages, as usual. In addition, there are new messages for requesting and granting the actual required resources. We want to keep the resources manipulated by the two different algorithms conceptually separate, so imagine *duplicates* of the ones needed by the dining algorithm. Call these *bottles* to distinguish them from the forks.

11.3.3 Drinking Philosophers Algorithm

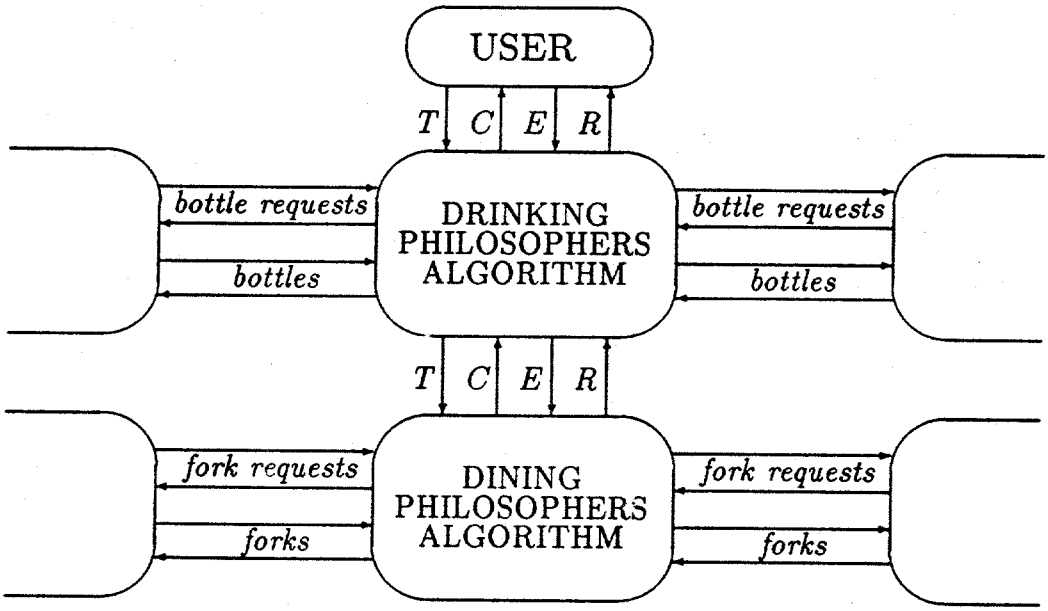


Figure 11.4: Proposed architecture for Lynch version of Chandy-Misra drinking philosophers algorithm.

While in R

- Satisfy all bottle requests.
- If the subroutine is in C, send it into E.

While in T

- Send requests for all bottles you need and do not have.
- If the subroutine is in (or reaches) R, send it into T. (This helps to give priority for bottles when the subroutine is in C.)
- If you have a request, defer it if you need the bottle and the subroutine is in C. Otherwise, satisfy it.
- Enter C when you have all of the bottles that you need.

While in C

- Satisfy all requests for bottles you do not need and defer requests for those you are using.
- If the subroutine is in C, send it into E.

While in E

- Satisfy all deferred requests (and any new requests).
- If subroutine is in C, send it into E.
- Proceed to R.

11.3.4 Correctness

Mutual exclusion

Follows from the fact that shared bottles are held by at most one process at a time, and that no required bottles are given up while in C.

Fairness

We will show that every thirsty philosopher drinks eventually.

Lemma 11.9 *If p_i is in T and its subroutine is in C, then eventually p_i reaches C.*

Proof: The subroutine stays in C until p_i advances. The subroutines of p_i's neighbors are not in C, (by the mutual exclusion property of the dining philosophers algorithm), so they will eventually grant the bottle requests. So eventually p_i gets all of its required bottles and proceeds to C. ■

Lemma 11.10 *If p_i's subroutine is in C, eventually this subroutine will proceed to E.*

Proof: If p_i is in C, E, or R, then p_i will send the subroutine to E explicitly. If p_i is in T and its subroutine is in C, then eventually p_i proceeds to C by Lemma 11.9. It then sends the subroutine to E. ■

This lemma means that the well-formedness assumptions made about the users of the dining philosophers algorithm are satisfied. This, in turn, implies that the dining philosophers subsystem gives the required liveness (no deadlock, no lockout) properties.

Theorem 11.11 *Every thirsty philosopher drinks eventually.*

Proof: If p_i is in T and its subroutine in E, then its subroutine eventually proceeds to R by the guarantee of *no lockout on dining philosopher requests.* If p_i is in T and its subroutine in R, then it sends its subroutine to T. If p_i is in T and its subroutine in T, then its subroutine eventually proceeds to C by the guarantee of *no lockout on dining philosopher requests.* Then, by Lemma 11.9, p_i reaches C. ■

11.4 Exercises

1. Give a correct I/O automaton for the Ricart–Agrawala distributed mutual exclusion strategy described informally in class. Your automaton definition should include the following: signature, state variables (with types and initial states), preconditions and effects for each locally controlled action, effects for each input action, and a partition of the output actions.

   (If you prefer, you can instead give an automaton for an improved version of the algorithm that uses the strategy of Carvalho et al.)

2. Give a generalized version of the Burns left–right alternating dining philosophers solution that works for an odd number of philosophers. Prove an upper bound independent of n (for the maximum time a philosopher must wait to eat after becoming hungry) for your algorithm.

3. Show that there exists an adversary for the Rabin–Lehmann randomized dining philosophers algorithm such that the probability of locking out a particular process is nonzero.

4. Consider the Chandy–Misra drinking philosophers algorithm using the Lynch dining philosophers subroutine. State and prove an upper bound on the time a philosopher must wait to drink after becoming thirsty.

*5. Recall that Lamport's Bakery algorithm for mutual exclusion used unbounded single-writer multi-reader *safe* registers. Suppose you were given the luxury of using unbounded single-writer multi-reader *regular* registers. Give a simplified (a few lines of code) version of the algorithm. Show that your algorithm has the same mutual exclusion, fairness, and failure resiliency[1] properties as Lamport's. This is another example of how a simple change to the underlying model of computation can make a drastic difference in the kinds of solutions obtainable.

*6. Give an appropriate "progress" condition that is applicable to as general a class of exclusion problems as possible. Verify that your condition is satisfied by Dijkstra's dining philosophers algorithm, Burns' dining philosophers algorithm, and the Chandy–Mistra drinking philosophers algorithm. Argue that your condition is strong enough; for example, your condition should rule out any dining philosophers algorithm that might not allow a philosopher to eat even though its neighbors are neither hungry nor eating.

[1] When a process fails, it goes to the remainder region and eventually all its shared variables are reset to zero.

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 12: October 25

Lecturer: Nancy Lynch *Scribe: Sanjay Ghemawat*

12.1 Concurrent Read/Write Registers

We now discuss the construction of registers which allow concurrent readers and writers. All the registers are modeled as I/O automata as in Figure 12.1. The index i on the read

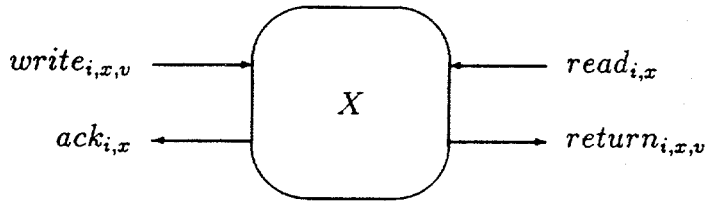


Figure 12.1: Concurrent Read/Write Register X

and write operations corresponds to a process calling the register. That is, for each process i that can read the register X, X has a $read_{i,x}$ operation (similarly for write operations). Therefore, from the point of view of the register, each index can be regarded as just naming an input line. We assume that operations on each line are invoked sequentially, i.e., no new operations are invoked on a line until all previous operations invoked on that line have returned. But otherwise, operations can overlap.

12.1.1 Register Types

Only single writer registers are considered in the following discussion. Because of this restriction, writes never overlap one another. Overlapping reads are assumed not to affect one another, so we only need to consider the case of a read operation overlapping one or more write operations. There are three different possibilities in this case. The weakest possibility is a *safe register* in which a read that is overlapping a write can return an arbitrary value. The strongest possibility, an *atomic register*, was defined in Lecture 3. The other possibility,

a *regular register* falls somewhere in between safe and atomic registers. A read operation on a regular register returns the correct value if it does not overlap any write operations. However, if a read overlaps one or more writes, it has to return the value of the register either before or after any of the writes it overlaps. For example, consider the two read operations in

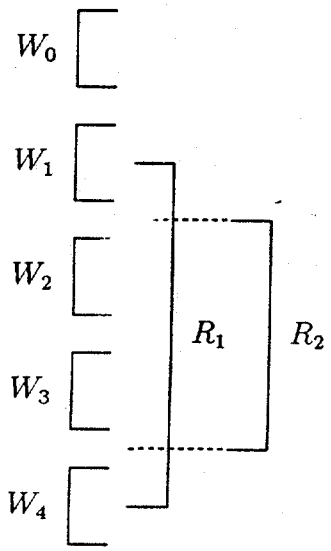


Figure 12.2: Read Overlapping Writes

Figure 12.2. The set of *feasible writes* for R_1 is $\{W_0, W_1, W_2, W_3, W_4\}$ because it is allowed to return a value written by any of these write operations. Similarly, the set of feasible writes for R_2 is $\{W_1, W_2, W_3\}$. The reader should note that there need not be any relation between the value returned by R_1 and the value returned by R_2. It should also be noted that only atomic registers can have multiple writers.

12.2 Implementation Relationships for Registers

In the following discussion (adapted from [Lamport86]), binary valued registers are distinguished from multiple valued (k-ary) registers and single reader registers from n-reader registers. We consider the twelve different kinds of registers this classification gives rise to, and see which register types can be used to implement other types. In the following diagrams, an arrow from register type A to register type B signifies that B can be implemented using A. The implementation relationships in Figure 12.3 should be obvious.

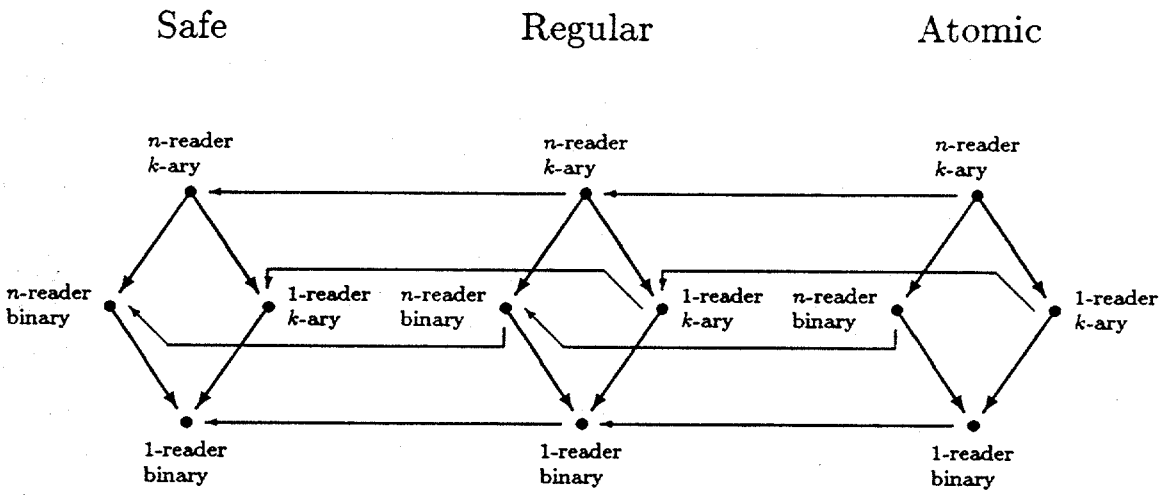


Figure 12.3: Obvious implementation relationships between register types. An arrow from type A to type B means that B can be implemented using A

12.3 Register Constructions

Lamport presents five constructions to show other implementation relationships. All of these constructions have a similar flavor. For example, consider Figure 12.4. Two 1-reader registers are being used to implement a 2-reader register. The 1-reader registers are called the *physical* registers, and the 2-reader register is called the *logical* register. In all of the following constructions, a logical register is constructed from one or more physical registers. Each input line to the logical register is connected to a process. These processes in turn are connected to one or more of the physical registers using internal lines. Exactly one process is connected to any internal line of a physical register. This guarantees that operations on each internal line are invoked sequentially. Processes connected to external write lines are called write processes, and processes connected to external read lines are called read processes. Note that nothing prevents a read process from being connected to an internal write line of a physical register. Given this background, the construction process can be stated in the following manner — *if the physical registers satisfy their specification and operations from outside are invoked sequentially on each line, then the composed system's fair behaviors all satisfy the specification for the logical register.*

In the following constructions, actions on external lines are always specified in upper-case, whereas actions on internal lines are specified in lower-case. For example, WRITE and READ denote external operations whereas *write* and *read* denote internal operations.

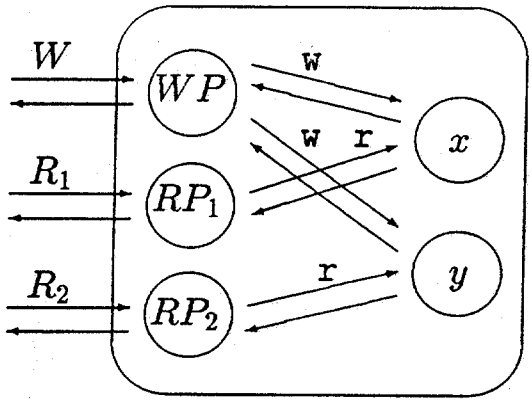


Figure 12.4: Example: Implementing a 2-reader register with two 1-reader registers

12.3.1 N-Reader Registers from 1-Reader Registers

The following construction implements an n-reader safe register from n 1-reader safe registers, and an n-reader regular register from n 1-reader regular registers. The write process is connected to the write lines of all n internal registers as in Figure 12.5. Read process i is connected to the read line of the ith physical register.

Code for WRITE(v)

For all i in $\{1, \ldots, n\}$, send *write*(v) to x_i. Wait for *acks* from all x_i and then send ACK. The *writes* can be done either concurrently, or in any order.

Code for READ$_i$

Send *read* to x_i. Wait for *return*(v) and then send RETURN(v).

Claim 12.1 *If $x_1, \ldots, x_n$ are safe registers, then so is the logical register.*

Proof: Within each WRITE, for any particular x_i, exactly one *write* is performed on that register. Therefore, since WRITE operations occur sequentially, *write* operations for a particular x_i are also sequential. In addition, *read* operations for a particular x_i are also sequential. Therefore, each physical register has the required sequentiality of accesses.

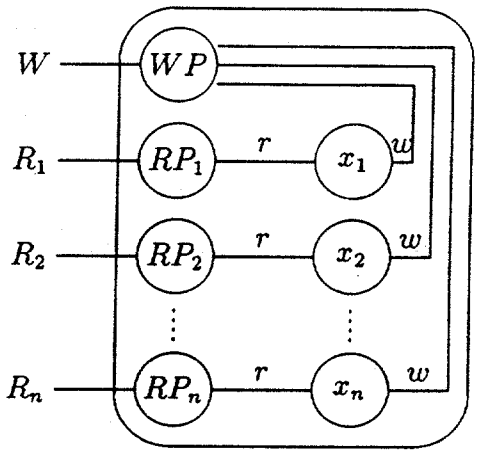


Figure 12.5: N-Reader Registers from n 1-Reader Registers

If a READ, say by RP_i, does not overlap any WRITE, then its contained *read* does not overlap any *write* to x_i. Therefore, safety of x_i assures that the *read* operation gets the value written by the last completed *write* to x_i. This is the same value as written by the last completed WRITE, and since RP_i returns this value, this READ returns the value of the last completed WRITE. ∎

Claim 12.2 *If $x_1, \ldots, x_n$ are regular registers, then so is the logical register.*

Proof: We can reuse the preceding proof to get the required sequentiality of accesses to each physical register, and to prove that any READ which does not overlap a WRITE returns the correct value. Therefore, we only need to show that if a READ R overlaps some WRITE operations, then it returns the value written by a feasible WRITE. Since the *read* r for R falls somewhere inside the duration of R, the set of feasible *writes* for r corresponds to a subset of the feasible WRITES for R. Therefore, regularity of the physical register x_i implies that R gets one of the values written by the set of feasible WRITES. ∎

Claim 12.3 *This construction does not make the logical register atomic even if the x_i are atomic.*

With this construction, Figure 12.3 reduces to Figure 12.6.

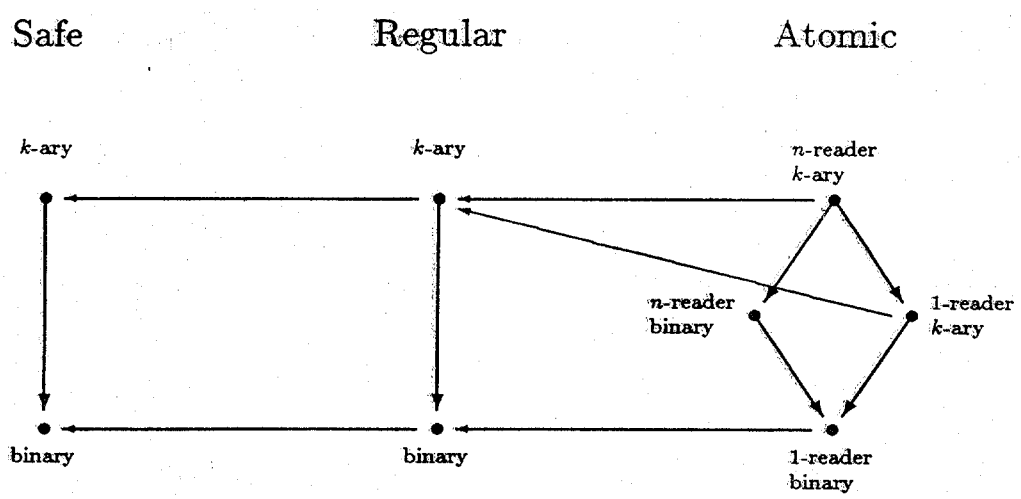


Figure 12.6: Collapsed Implementation Relationships

12.3.2 Wait-Free Registers

The previous construction guarantees that all logical operations terminate in a bounded number of steps of the given process, regardless of what the other processes do. This is a general property we would like for all such constructions. Wait-freeness can either be formulated in terms of a bounded number of the process's own steps for operations, or in terms of a time bound on operations, given that the physical registers have a very fast response, and that the process's step time is bounded. This property is important, because registers obeying it allow non-delayed access to shared memory.
It would be useful to give a more careful definition of wait-freeness.

12.3.3 K-ary Safe Registers from Binary Safe Registers

If $k = \lfloor 2^l \rfloor$, then we can implement a k-ary safe register using l binary safe registers. We do this by storing the ith bit of the value in binary register x_i. The logical register will allow the same number of readers as the physical registers do (see Figure 12.7).

Code for WRITE(v)

For i in $\{1, \ldots, l\}$ (any ordering), write bit i of the value to register x_i.

Code for READ$_i$

For i in $\{1, \ldots, l\}$ (any ordering), read bit i of value v from register x_i. RETURN(v).

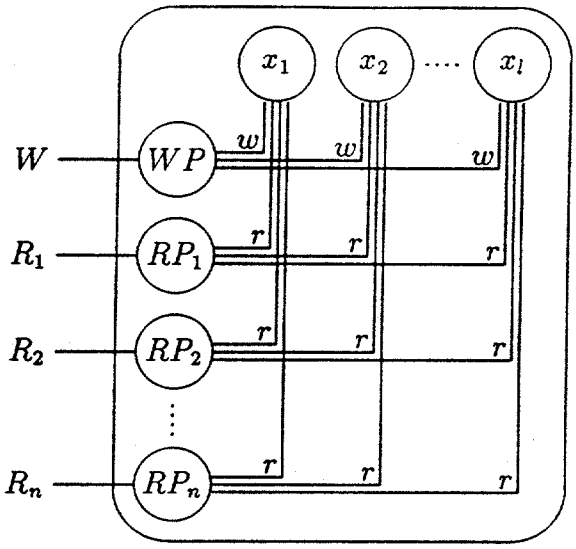


Figure 12.7: K-ary Safe Registers from Binary Safe Registers

Note that unlike the first construction, this construction works only for safe registers, i.e., a k-ary regular register cannot be constructed from a binary regular register using this method. With this construction, Figure 12.6 reduces to Figure 12.8.

12.3.4 Binary Regular Register from Binary Safe Register

A binary regular register can easily be implemented using just one binary safe register (see Figure 12.9). The basic idea is that the write process WP, locally keeps track of the contents of register x (this is easy because WP is the only writer of x). WP does a low-level *write* only when it gets a WRITE which would actually change the value of the register. If the WRITE is just rewriting the old value, then WP finishes the operation right away without touching x. Therefore, all low level *writes* toggle the value of the register. Now consider the case when a READ is overlapped by a WRITE. If the corresponding *write* is not performed (i.e., value is unchanged), then register x will just return the old value, and this READ will be correct. If the corresponding *write* is performed, x may return either 0 or 1. However, both 0 and 1 are in the feasible value set of this READ because the overlapping WRITE is toggling the value of the register. Therefore, the READ will be correct. Figure 12.8 now reduces to Figure 12.10.

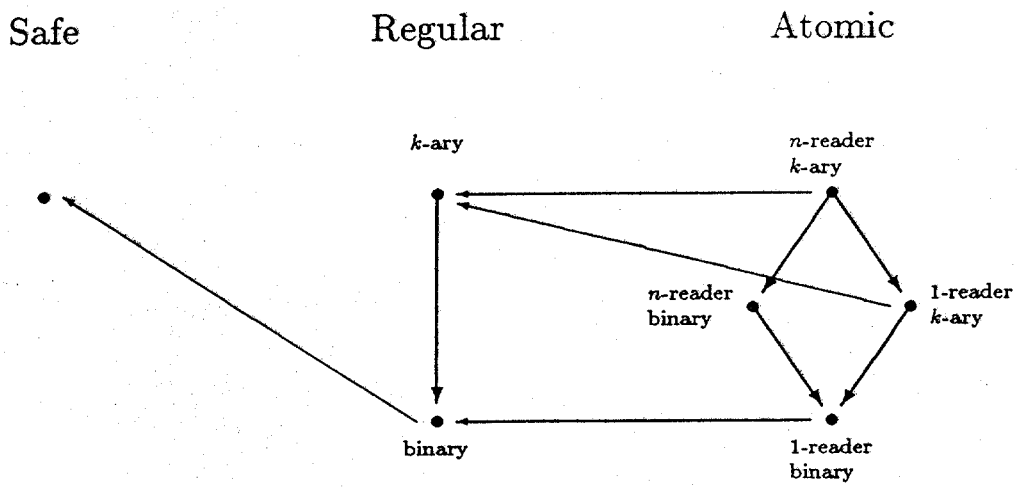


Figure 12.8: Collapsed Implementation Relationships

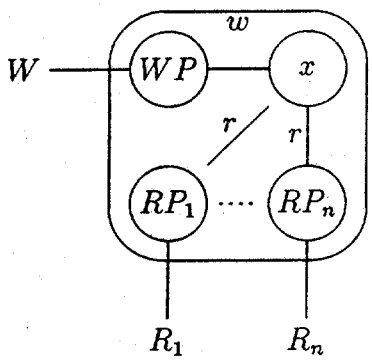


Figure 12.9: Binary Regular Register from Binary Safe Register

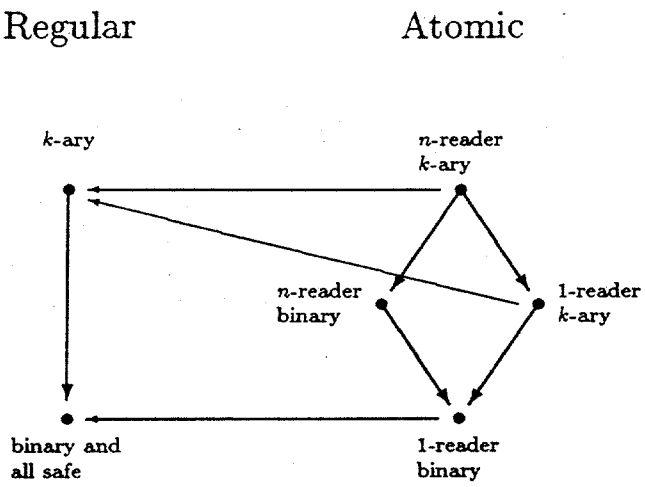


Figure 12.10: Collapsed Implementation Relationships

12.3.5 K-ary Regular Register from Binary Regular Register

We can implement a k-ary regular register using k binary regular registers as in Figure 12.11. If the initial value of the register is v_0, initially x_{v_0} is 1 and other physical registers are all 0.

Code for WRITE(v)

write 1 to x_v. Then, in order, *write* 0 to $x_{v-1}, \ldots, x_0$.

Code for READ

read $x_0, x_1, \ldots, x_{k-1}$ in order until some $x_v = 1$ is found. RETURN(v).

RP_i is guaranteed to find a non-zero x_v because whenever a physical register is zeroed out, there is already a 1 written in a higher index register.

Claim 12.4 *If a* READ R *sees a* 1 *in* x_v, *then* v *must have been written by a* WRITE *which is feasible for* R.

Proof: Suppose not. Then R sees $x_v = 1$, and neither an overlapping or immediately preceding WRITE wrote v to the logical register. Then v was written either sometime in the past, or $v = v_0$ (initial value). For the moment, ignore the initial value case. Since v is written by W_1 (which is not a feasible write for R), there must be a W_2 completely after W_1 which completely precedes R (otherwise W_1 would be a feasible WRITE for R). This W_2 must write something $< v$ because if it wrote a value $> v$, it would set $x_v = 0$ before R could

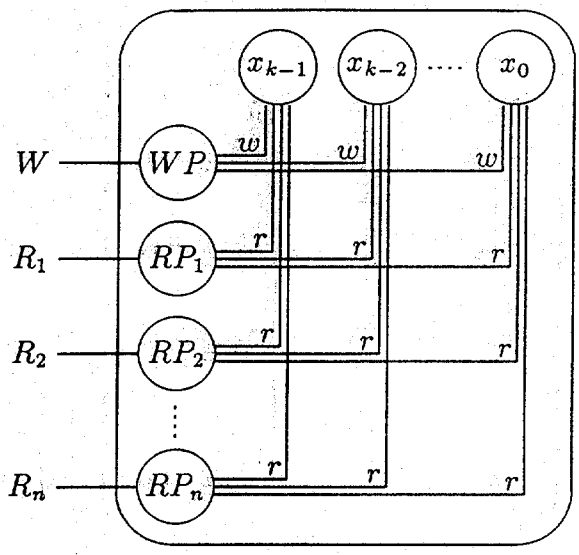


Figure 12.11: K-ary Regular Registers from Binary Regular Registers

see $x_v = 1$, and if it wrote a value $= v$, it would reset $x_v = 1$, but then R would not get the result of W_1, but of W_2. So, let v' be the biggest value (must be $< v$) such that a *write*(1) to $x_{v'}$ completely follows W_1 and completely precedes the *read* of $x_{v'}$ in R. Since R reads the registers in order $x_0, \ldots, x_{k-1}$, and it returns value $v > v'$, R must have seen $x_{v'} = 0$. But, v' was set to 1 sometime before *read* of $x_{v'}$ in R. Therefore, there exists some *write*(0) to $x_{v'}$ which follows the *write*(1) to $x_{v'}$ and either precedes or overlaps the *read* of v' in R. But this can only happen if there is an intervening *write*(1) to some $x_{v''}$ such that $v'' > v'$. This is a contradiction to the definition of v' being the biggest such value. Note the *write*(1) to $x_{v''}$ completely precedes the *read* of $x_{v''}$ in R because the *write*(0) to $x_{v'}$ either precedes or overlaps the *read* of $x_{v'}$ in R, and $v'' > v'$.
Note: The case when v is the initial value can be treated similarly. ■
The implementation replationships now collapse as shown in Figure 12.12.

12.3.6 1-Reader K-ary Atomic Register from Regular Register

It is possible to construct a 1-writer, 1-reader k-ary atomic register from two 1-reader regular registers as in Figure 12.13. Regular register x stores tuples of form $\langle \text{old}, \text{new}, \text{num}, \text{color} \rangle$ where $\text{old}, \text{new} \in V$, $\text{num} \in \{1, 2, 3\}$ and $\text{color} \in \{\text{red}, \text{blue}\}$. Register y stores colors from $\{\text{red}, \text{blue}\}$.

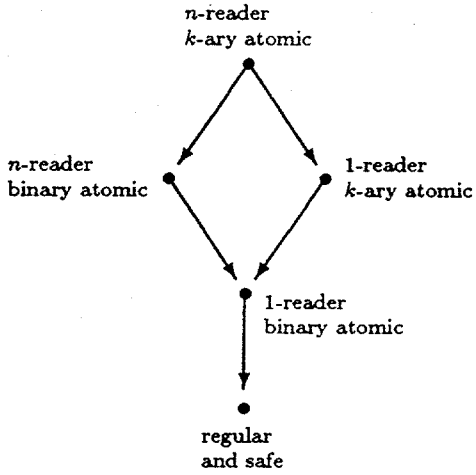


Figure 12.12: Collapsed Implementation Relationships

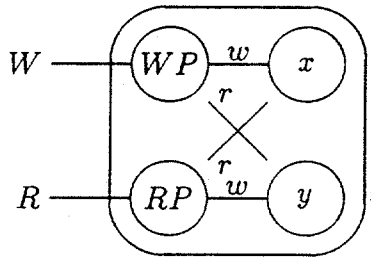


Figure 12.13: 1-Reader Atomic Register from Regular Registers

Code for WRITE(v)

Remember value of x locally as x.new.
newcolor $\leftarrow \neg(read\ y)$
oldvalue $\leftarrow x$.new
for $i = 1, 2, 3$ **do** *write* $\langle$oldvalue$, v, i,$ newcolor$\rangle$ to x.

Code for READ

Remember last two reads in x' and x''.
$x'' \leftarrow x'$
$x' \leftarrow read\ x$
write x.color to y.
case
num $= 3$:
new-returned $\leftarrow$ true
RETURN x'.new
x'.num < 3 and x'.num $\geq (x''$.num $- 1)$ and new-returned and x'.color $= x''$.color:
RETURN x'.new
else:
new-returned $\leftarrow$ false
RETURN x'.old
end case

Read [Lamport86] for the correctness proof. The implementation relations now collapse as in Figure 12.14. For other constructions (including multi-writer atomic registers), see

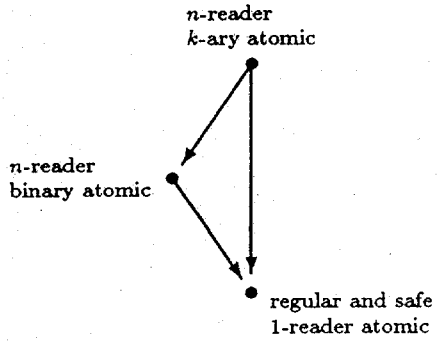


Figure 12.14: Collapsed Implementation Relationships

[Bloom87], [BurnsP87], [NewmanW87,] [SinghAG87], [VitanyiA86] and Lecture 13.

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 13: October 27

Lecturer: Nancy A. Lynch *Scribe: Christopher P. Colby*

13.1 Multi-writer Registers

In this lecture we continue our discussion of register constructions. In particular, we will consider constructions of multi-writer registers from single-writer registers. Bloom developed a construction that yields 2-writer n-reader registers from 1-writer $n+1$-reader registers. Vitanyi-Awerbuch developed an n-writer n-reader construction from 1-writer 1-reader registers, but the 1-writer registers must be of unbounded size. Peterson-Burns-Schaffer developed a construction of an n-writer n-reader register from 1-writer 1-reader registers of unbounded size.

13.2 Bloom's 2-writer Construction

Bloom [Bloom87] developed a method for constructing a 2-writer n-reader atomic register from two 1-writer $n+1$-reader atomic registers. As you will recall, atomic registers are registers which act as if reads and writes do not overlap. Each read and write operation is effectively shrunk to a point in the interval between the start and end of the operation. In reasoning about Bloom's construction, one can assume that the reads and writes of the two 1-writer atomic registers occur as single points in the execution. That very fact is itself an example of why atomic registers are often desirable.

The construction of the 2-writer atomic register is shown in Figure 13.1. It consists of two atomic 1-writer $n+1$-reader registers (x_0 and x_1), n read processes ($\mathrm{RP}_1 \ldots \mathrm{RP}_n$), and two write processes (WP_0 and WP_1). Each atomic register x_i holds a pair $(value, tag)$, where *value* is the value of the logical register and *tag* is either 0 or 1. Initially, x_0 and x_1 hold $(v_{init}, 0)$, where v_{init} is the initial value of the logical register. The algorithms for the read processes and the write processes are shown in Figure 13.2. They are modeled as I/O Automata.

13.2.1 Correct behavior

A well-formed behavior of this system is one in which no invocation to WP_0, WP_1, or any RP_i occurs until the previous invocation to the same write or read process has terminated.

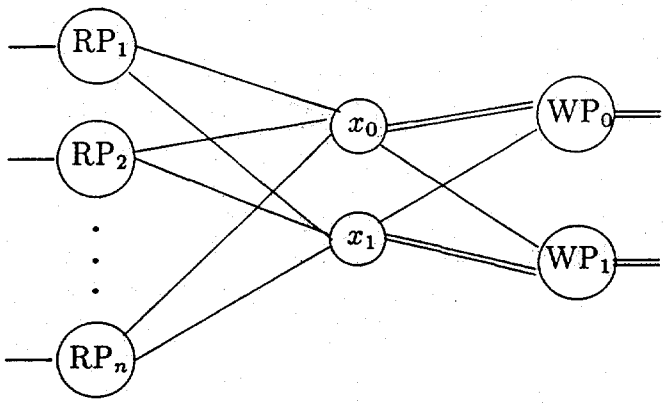


Figure 13.1: Architecture of Bloom's 2-writer atomic register construction. Lines between processes and registers denote access channels. Read access channels are shown as single lines and write access channels as double lines.

Algorithm for WP_i (writing a value v):

read (v', t') from $x_{\neg i}$
$t \leftarrow (i \oplus t')$
write (v, t) to x_i

Algorithm for RP_i:

read (v_0, t_0) from x_0
read (v_1, t_1) from x_1
$r \leftarrow t_0 \oplus t_1$
read (v_2, t_2) from x_r
return v_2

Figure 13.2: Algorithms for the read and write processes.

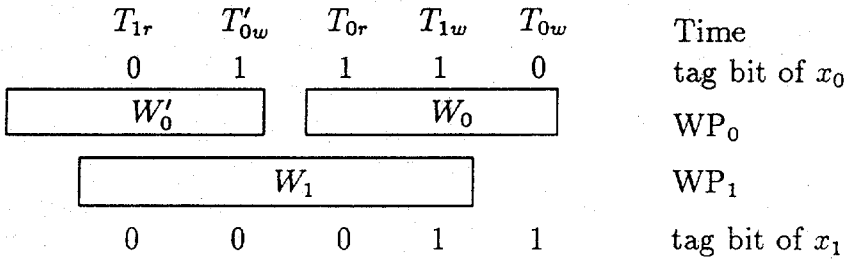


Figure 13.3: The obliterator of an impotent WRITE is potent.

The constructed 2-writer register is atomic. Thus, for any well-formed fair behavior α of this system, the read and write operations on this logical register (henceforth called READs and WRITEs) must be able to "shrink to a point". This behavior α arises from some fair schedule β of the system. Because the physical registers x_0 and x_1 are atomic, β can be extended by inserting a star ($*$) for each read or write from or to those registers such that the sequence of these $*$-*actions* for each physical register is a correct serial behavior for read/write invocations of that register with the given initial value. Let γ be this extended sequence. We will augment γ by adding $*$-actions for each READ or WRITE of the *logical* register and then argue that the result is a correct logical register behavior. To say that a WRITE or READ "occurs at time T" means that its $*$-action occurs at time T in the schedule.

For a given WRITE operation, we refer to the resulting low-level write to x_0 or x_1 as the *contained write.* Likewise, any low-level read operation within a READ or WRITE is called a *contained read.*

A WRITE is one of two types: *potent* or *impotent.*

Definition A WRITE W by WP_i is *potent* iff the modulo 2 sum of the tag bits of x_0 and x_1 immediately after the $*$-action of the contained write is i. Otherwise, it is *impotent.*

Definition a WRITE W_0 is *obliterated* by WRITE W_1 if the contained write of W_1 occurs between the contained read and write of W_0 and if W_1 is the last such WRITE.

Lemma 13.1 *Every impotent WRITE is obliterated by precisely one WRITE.*

Proof: By the definition of obliteration, there cannot be more than one obliterator of any WRITE. Let W_0 be a WRITE by WP_0 which is not obliterated by any other WRITE. Then, no writes to x_1 occur between the read of W_0 and the write of W_0. So, the tag bit of x_1 immediately after the write of W_0 is the same as it was during the read of W_0. W_0 chooses to write x_0 with a tag bit equal to that bit of x_1. So, since W_1 cannot write to register x_0, the tag bits are equal right after the write of W_0. Thus, W_0 is potent. The same reasoning follows for WRITEs by WP_1. Therefore, any WRITE not obliterated by another WRITE is potent. ∎

Lemma 13.2 *The obliterator of an impotent WRITE is potent.*

Proof: Assume the contrary. Let W_0 be the first impotent WRITE with an impotent obliterator. Let W_1 be its obliterator. W_1 is impotent, so according to Lemma 13.1 it must have its own obliterator. Let W_0' be the obliterator of W_1. Figure 13.3 shows the sequence of internal reads and writes for W_0', W_1, and W_0. Those reads and writes are labelled with times immediately after their $*$-actions. The $*$-actions of the internal reads occur at times T_{0r}' (not shown because of irrelevance), T_{1r}, and T_{0r}. The $*$-actions of the internal writes occur at

times T'_{0w}, T_{1w}, and T_{0w}. Obviously, $T'_{0w} < T_{0r} < T_{0w}$. By the definition of obliteration, $T_{1r} < T'_{0w} < T_{1w}$ and $T_{0r} < T_{1w} < T_{0w}$. This yields the complete ordering, shown in Figure 13.3, $T_{1r} < T'_{0w} < T_{0r} < T_{1w} < T_{0w}$.

Call x_i's tag bit t_i. Assume, without loss of generality, that $t_0 = 0$ at T_{0w}. Since W_0 is impotent, $t_1 = 1$ at T_{0w}. Since W_0 wrote t_0 to be 0, it must have read $t_1 = 0$ at time T_{0r}. Since the only write to t_1 occurs at T_{1w}, $t_1 = 0$ at all times before T_{1w} and $t_1 = 1$ at T_{1w}. Since W_1 wrote t_1 to be 1, it must have read $t_0 = 0$ at time T_{1r}. Since W_1 is impotent, t_0 must be 1 at T_{1w}. Since the last write to t_0 before T_{0w} occurs at T'_{0w}, $t_0 = 1$ at T'_{0w} and T_{0r}.

At time T'_{0w}, $t_0 = 1$ and $t_1 = 0$. Thus, W'_0 is impotent, and W_1 is an impotent write with an impotent obliterator. However, W_0 was stated to be the first impotent write with an impotent obliterator. This contradiction establishes the proof. ■

13.2.2 Insertion of $*$-actions

With Lemma 13.1 and 13.2, we can now proceed to insert the $*$-actions for the WRITEs and READs into into any behavior. First, the $*$-actions for all WRITEs are inserted simultaneously. Then, the $*$-actions for all READs of potent WRITEs are inserted. Then, the $*$-actions for all READs of impotent WRITEs are inserted. Finally, READs of the initial value are handled. All $*$-actions must lie within the corresponding READ or WRITE.

First, insert the $*$-actions of the WRITEs. If the WRITE is potent, insert the $*$ immediately after the $*$-action of its contained write. If the WRITE is impotent, insert the $*$ immediately before the $*$-action of the WRITE that obliterated it. It is clear that the $*$-actions of potent WRITEs lie within the WRITE interval. By Lemma 13.2, obliterators of impotent WRITEs are potent. Thus, the $*$-action of the obliterator of an impotent WRITE lies within the interval for that impotent WRITE. Therefore, the $*$-action of the impotent WRITE also lies within that interval.

Now, insert the $*$-actions for READs of potent WRITEs. Place the $*$ immediately after the later of the $*$-action of the WRITE being read and the first contained read of the READ. It is obvious that the $*$ falls within the READ interval.

Lemma 13.3 *If R is a READ of a potent WRITE W, then the $*$-action of W is the last $*$-action of any WRITE preceding the $*$-action of R.*

Proof: Let T_w be the time of the contained write of W. Let T_0, T_1, and T_2 be the times of the contained reads of R. There are two cases depending on the ordering of T_w and T_0. The first case is that $T_w > T_0$. In that case, the $*$-action of R occurs immediately after the $*$-action of W, with only $*$-actions of other READs possibly intervening. The Lemma is clearly true in this case.

The other case is that $T_w < T_0$. Assume, without loss of generality, that WP_0 is the writer of W. Since R reads W, WP_0 did not perform a real write between T_w and T_2. Therefore, no

$*$-actions of potent WRITEs occurred in that interval. If a $*$-action of in impotent WRITE occurred in that interval, then so would the $*$-action of its (potent) obliterator. Thus, no $*$-actions of WRITEs by WP_0 occur between T_w and T_2.

Now consider the case where a $*$-action of a WRITE W_1 by WP_1 occurs between T_w and T_2. If W_1 were impotent, then the $*$-action of W_1's obliterator (a WRITE by WP_0) would also occur between T_w and T_2. As shown in the previous paragraph, this is not possible. Thus, W_1 must be potent. Assume, without loss of generality, that W set the tag of x_0 to 0 at T_w. Since W is potent, the tag of $x_1 = 0$ at T_w, also. Therefore, W_1 must set the tag of x_1 to 1. Since W_1 is potent, this occurs before T_0. Thus, the tag of $x_0 = 0$ and the tag of $x_1 = 1$ between the $*$-action of W_1 and T_2. Therefore, R reads 0 and 1 at T_0 and T_1, respectively, and reads from x_1 at T_2. However, this is a contradiction, since R reads from x_0 at T_2. ∎

Now, insert the $*$-actions for READs of impotent WRITEs. Place the $*$ immediately after the $*$-action of the WRITE. A statement analogous to Lemma 13.3 clearly applies here, but it is not as obvious that the inserted $*$ lies within the interval of the READ.

Lemma 13.4 *If R is a READ of an impotent WRITE W, then the $*$-action of W occurs within the interval of R.*

Proof: Since R reads W, the contained write of W must precede the last (third) contained read of R. Since $*$-actions of impotent WRITEs precede their contained write, the $*$-action of W clearly cannot occur after the interval of R.

If the $*$-action of W occurs before the interval of R, then the $*$-action of its (potent) obliterator occurs between it and the beginning of the interval of R. Assume, without loss of generality, that WP_0 performs W. Let W_1 be the obliterator of W. Since W_1 is potent, the tag bits immediately after its $*$-action sum to 1. Since W is impotent, the sum of tag bits immediately after its contained write is also 1. Since R reads W, WP_0 does not perform any real writes between the $*$-action of W and the final contained read of R. Thus, the tag bit of x_0 remains unchanged in that interval. WP_1 may perform real writes, but all of them will use the same tag bit for x_1. Thus, the sum of the tag bits remains unchanged from the $*$-action of W_1 to the last contained read of R. As stated above, this sum is 1. But then R would read the sum of the tag bits as 1 and thus read from x_1. This is a contradiction, since we assumed above that WP_0 performs W. ∎

Finally, insert the $*$-actions for READs of the initial value, by placing them immediately after the second contained read. Let R be a READ of the initial value. Clearly, the $*$-action of R is within the interval of R. It then suffices to show that no $*$-action of any WRITE occurs before R's second read. If R read from x_1 on its third contained read, then it must of read different tags during its first read (of x_0) and second read (of x_1), and, since both tags are initially 0, R must not have read the initial value. Thus, R read from x_0 on its third contained read. So, there are no writes by WP_0 before R's third read. If there is a write by WP_1 before R's second read, then R would have read different tags during its first and

Notation: There are n writers, $WP_1, \ldots, WP_n$. There are n readers, $RP_1, \ldots, RP_n$.

Shared variables: There are $4n^2$ 1-writer/1-reader atomic variables, $x_{1,1}, \ldots, x_{2n,2n}$. WP_i reads $x_{i,1}, \ldots, x_{i,2n}$ and writes $x_{1,i}, \ldots, x_{2n,i}$. RP_i reads $x_{i+n,1}, \ldots, x_{i+n,2n}$ and writes $x_{1,i+n}, \ldots, x_{2n,i+n}$. The value that each of these holds is a tuple (*value, tag*), where *tag* is a tuple (*version, j*), where *version* $\in \{0, \ldots\}$, initially 0, and $j \in \{1, \ldots, n\}$.

Algorithm for WP_i to write the value *new*: Read $x_{i,1}, \ldots, x_{i,2n}$ in any order. Determine the greatest *tag* (t, j). Write $(new, (t+1, i))$ into $x_{1,i}, \ldots, x_{2n,i}$ in any order.

Algorithm for RP_i: Read $x_{i+n,1}, \ldots, x_{i+n,2n}$ in any order. Determine the $(v, (t, j))$ with the greatest *tag*. Write $(v, (t, j))$ into $x_{1,i+n}, \ldots, x_{2n,i+n}$ in any order.

Figure 13.4: The Vitanyi-Awerbuch n-writer register construction.

second read and thus would have read from x_1 during its third read. This is a contradiction. Therefore, no $*$-action of any WRITE occurs before the second contained read of any READ of the initial value.

So, $*$-actions may be inserted for every READ and WRITE in any fair behavior of the constructed 2-writer register such that each READ returns the value of the WRITE with the last $*$-action of any WRITE preceding the READ's $*$-action. It is thus a correct implementation of an atomic register.

13.3 Vitanyi-Awerbuch's n-writer Construction

Vitanyi-Awerbuch [VitanyiA86] developed a construction of an n-writer n-reader register from 1-writer 1-reader registers, but the 1-writer registers must be unbounded in size. Figure 13.4 describes the construction and the algorithms of the read and write processes. In Lecture 14, we will discuss its correctness.

13.4 Exercises

1. Consider Lamport's Construction 4, the one that shows how to implement $k - ary$ regular registers using binary regular registers.

   Show that even if the binary registers are atomic, the resulting $k - ary$ register is not atomic.

2. Define carefully what it means for one kind of register to be capable of implementing another in a wait-free manner. Sketch why your notion is transitive.

3. In Bloom's 2-writer algorithm, indicate why the third read within the READ protocol is necessary; i.e., what goes wrong if the READ just returns the value already read (in the first or second read) from the appropriate register?

4. Suppose that the Hirschberg-Sinclair leader election algorithm is modified so that successive powers of k are used for path lengths, $k > 2$, instead of successive powers of 2. Analyze the time and message complexity of the modified algorithm, similarly to the way the original algorithm was analyzed in class. Compare the results to those for the original algorithm.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 14: November 1

Lecturer: Nancy A. Lynch | *Scribe: Mike Eisenberg*

14.1 nWnR Register Construction (Cont.)

In Lecture 13, we presented the Vitanyi-Awerbuch n-writer, n-reader register construction. The following lemma provides us with an opportunity to do "higher-order" reasoning about the construction:

Definition A sequence β of external actions of a logical register is *regular* if each read process and write process satisfies variable well-formedness and if every invocation has a corresponding later response.

Lemma 14.1 *Suppose a regular sequence β has a partial ordering $<$ of all operations (paired invocations and responses) such that:*

- *If end_i precedes $begin_j$, then it cannot be the case that $operation_j < operation_i$.*
- *$<$ orders all WRITE operations, and orders all READ operations with respect to all WRITE operations.*
- *The value returned by each READ operation is the value written by the last WRITE operation ordered before it in $<$ (or the initial value, if no WRITES occur before the READ according to $<$).*

Then β is a correct atomic register behavior.

The first condition in the lemma requires a consistent ordering for non-overlapping intervals – that is, if some operation ends before another begins, the ordering must place the first operation before the second. The second and third conditions are used to show that any total ordering of the operations consistent with $<$ is a correct serial behavior of the read-write register.

A consequence of this lemma is that we can show a register is atomic by showing that:

1. The register preserves well-formedness;
2. In a fair behavior, a return occurs for every invocation; and
3. A partial ordering like the one in the lemma exists for all fair behaviors of the register.

We now prove the lemma.

Proof: We specify, for each READ and WRITE interval, a corresponding "star event" (labelled $*$) somewhere within that interval such that every READ gets the value written by the WRITE whose $*$ immediately precedes the READ's $*$. The rule is simple: we insert the $*$ for event i (denoted by $*_i$) immediately after the latest among begin_i and all begin_j such that $\text{operation}_j < \text{operation}_i$. This will be specific enough so that we know that a $*$ can be placed within a given space; if more than one $*$ has to be placed within that space, we order them so that they are consistent with the partial ordering $<$.

Now, it must be the case that each $*_i$ is between begin_i and end_i. It is clearly after begin_i by the construction; and if it were to come after end_i, then it must be the case that end_i precedes begin_j, where $\text{operation}_j < \text{operation}_i$. However, this contradicts the first condition on the partial order; so we conclude that $*_i$ is correctly between begin_i and end_i.

We next show that the *precedes* order on $*$ symbols is consistent with the $<$ order on operations: that is, if $\text{operation}_j < \text{operation}_i$, then $*_j$ precedes $*_i$. Let $\text{operation}_j < \text{operation}_i$. We know that:

1. $*_j$ occurs after the last of begin_j and all begin_k where $\text{operation}_k < \text{operation}_j$.

2. $*_i$ occurs after last of begin_i and all begin_k where $\text{operation}_k < \text{operation}_i$.

This means that if $\text{operation}_j < \text{operation}_i$, then $*_i$ must occur after $*_j$, since $*_i$ is placed after the beginnings of all operations $<$ operation_i (this includes operation_j); and this set of operations will include the set that determines the placement of $*_j$ (which must therefore come before $*_i$). Thus the "precedes" order on $*$ must be consistent with the $<$ order on operations.

Finally, we claim that each READ operation must get the value of the last WRITE operation whose $*$ immediately precedes that of the given READ (or the *initial* value if there is no preceding WRITE). To prove this, we note that by the third property of the $<$ ordering, each READ gets the value of the last WRITE ordered before it by $<$. By the second property, we know that $<$ orders all READs relative to all WRITEs, and all WRITEs relative to each other. Since the precedes order is consistent with the $<$ order, a READ must get the value of the last preceding WRITE. ∎

Now that we have this lemma, it becomes a bit easier to think about the original Vitanyi-Awerbuch algorithm. If we can show that the operations in the algorithm can be ordered in such a way that the conditions of our lemma are satisfied, we have proved that the Vitanyi-Awerbuch algorithm correctly implements atomic register behavior. This leads us to the following claim:

Claim 14.2 *The Vitanyi-Awerbuch algorithm has an ordering among READ and WRITE operations that satisfies the conditions of Lemma 14.1.*

Proof: To prove our claim, we want to look back at the algorithm and order all significant operations in the way indicated by the lemma: that is, we want to order all WRITEs with respect to each other, and all READs with respect to all WRITEs. The method of ordering that seems intuitively plausible (and that does in fact work) is to order WRITE operations according to their associated tags, since there is a total order of WRITE operations based on tags. As for a READ operation, we can order it after the WRITE operation whose tag value that READ operation read. If there are several READs that get the same tag value, we don't really care, since we don't have to order READs relative to each other — only with respect to the WRITEs.

This ordering certainly satisfies the second condition of the lemma. As for the first condition, we want to show that if end_i precedes begin_j, then we can't have $\text{operation}_j < \text{operation}_i$. If operation_j and operation_i are both READs or both WRITEs, then the proof of this condition follows from the fact that the tag value at any given position in the table of variables can never decrease; if one WRITE finishes before another begins, the second WRITE must increase the tag value in its column beyond the tag value of the first WRITE, and will thus be ordered by $<$ after the earlier complete WRITE. (Similar observations apply to two consecutive complete READ operations, or a WRITE operation followed by a READ.) The only remaining case that we have to worry about is when operation_i is a READ operation and operation_j is a WRITE; but in this case, the tag associated with the WRITE will be bigger than that associated with the previously completed READ, so again we must have $\text{operation}_i < \text{operation}_j$. This completes the proof of the first condition.

Finally, to prove that the third condition is met, we need to show that the value returned by a READ is the value written by the last preceding WRITE operation in the $<$ ordering (or *initial*, if there are no preceding WRITEs). This follows immediately from our construction of the $<$ ordering: we placed each READ operation in this ordering precisely so that it would go after the WRITE operation whose corresponding value it read.

■

Now we have an interesting question to ponder: can we use a lemma similar to this one to provide a proof of Bloom's algorithm? It certainly isn't a straightforward task, since there are no obvious analogues in Bloom's algorithm to Vitanyi and Awerbuch's tags.

We might also consider trying to rephrase Bloom's algorithm in more abstract terms. In other words, it may be possible to describe Bloom's algorithm using higher-order concepts, and to show that the version of Bloom's algorithm that we used is merely an implementation of this more-abstractly-expressed version. The possibility of such a reworked version of Bloom's algorithm, or of other algorithms like the Singh-Anderson-Gouda construction (shown in Figure 14.1), is an open question.

Notation: The *writer* is denoted by W. There are m readers, $R_1, \ldots, R_{m-1}$. If x is a 1/1 atomic register, read(x) is a read operation that returns the value of x, and write(x, v) is a write operation that assigns v to be the value of x. Let $true = 1$ and $false = 0$.

Shared variables: All of these are either 1-writer/1-reader atomic registers that take on integer values, or banks (tuples) of such registers. Let x and y be arbitrary values.

WW is an integer written and read by W
Initially, $WW = x$
$\forall 0 \leq i < m$, WR_i is a tuple (old,new,seq_0,$\ldots$,seq_{m-1}), written by W and read by R_i
Initially, $WR_i = (y, x, 0, \ldots, true)$
$\forall 0 \leq i < m$, R_iW is an integer written by R_i and read by W
Initially, $R_iW = 0$
$\forall 0 \leq i, j < m$, R_iR_j is a tuple (old,new,seq_i,$flag$) written by R_i andread by W
Initially, $R_iR_j = (y, x, 0, true)$

Code for W to write the value new:

$old \leftarrow$read(WW)
if $old \neq new$ **then**
  for $k = 0, \ldots, k-1$ **do** [$seq_k \leftarrow$read(R_kW); $seq_k \leftarrow (seq_k + 1)$ mod 3]
  for $k = m-1, \ldots, 1$ **do** write(WR_k,(old,new,seq_0,$\ldots$,seq_{m-1},$false$))
  for $k = 1, \ldots, m-1$ **do** write(WR_k,(old,new,seq_0,$\ldots$,seq_{m-1},$true$))
  write(WW,new)

Code for R_i:

(old,new,seq_0,$\ldots$,seq_{m-1},$done$) $\leftarrow$read(WR_i)
write(R_iW,seq_i)
for $k = 0, \ldots, i-1$ **do** (old_k,new_k,seq'_k,$flag_k$) $\leftarrow$read(R_kR_i)
(old',new',seq'_0,$\ldots$,seq'_{m-1},$done'$) $\leftarrow$read(WR_i)

$p_i \leftarrow (old = old') \wedge (new = new') \wedge (\forall j : 0 \leq j < m : \ seq_j = seq'_j)$
for $k = 0, \ldots, i-1$ **do** $p_k \leftarrow p_i \wedge flag_k \wedge (old = old_k) \wedge (new = new_k) \wedge (seq_k = seq''_k)$
$flag \leftarrow p_0 \vee p_1 \vee \ldots \vee p_{i-1} \vee (p_i \wedge done \wedge done')$

for $k = i+1, \ldots, m-1$ **do** write(R_iR_k,(old',new',seq'_i,$flag$))
if $flag$ **then return**(new') **else return**(old')

Figure 14.1: Singh–Anderson–Gouda 1-writer/m-reader Atomic Register Construction

14.2 Herlihy Impossibility Result

Now that we have seen a variety of constructions that allow us to implement one type of register in terms of some other type, we might ask whether it is possible to find a wait-free implementation of higher-level register objects (such as atomic test-and-set registers) using atomic registers. Herlihy, Loui, and Abu-Amara showed — surprisingly — that it is in fact impossible to do this.

We will postpone the proof of their result until a later lecture — but, briefly, the essence of the proof is as follows: they consider a *distributed consensus problem* in which a group of processors start with different "votes" and have to reach agreement with a final vote. This problem has a known solution using atomic test-and-set registers; moreover, it can be solved in a way that is resilient to any number of stopping processes. (That is, no matter how many processes fail, the processes that continue operating will successfully reach a consensus.) They then show that this problem *cannot be solved* using any combination of atomic read-write registers. But if we could construct test-and-set registers out of atomic registers, then we could solve the consensus problem using read-write registers in such a way that the failure resiliency of the solution corresponds to the wait-free property of our construction. Thus, there can be no wait-free implementation of test-and-set registers using read-write registers.

14.3 Network Algorithms

The first section of this course was devoted to shared-memory algorithms. In this next few lectures we will discuss *network algorithms*, designed for groups of processes that communicate via messages.

Our discussion of network algorithms will focus on several representative problems: leader election algorithms, finding a minimum spanning tree, and taking global snapshots in a distributed system.

A loose classification of network algorithms separates them into two categories: *static* and *dynamic*. Static algorithms assume that inputs to the network are fixed. In other words, there are some number of processes arranged in a network communicating over edges (message buffers), and there are inputs set for each process at the beginning of the execution (no new inputs will come into the network during the course of the solution to the problem). The network produces some output to report the solution to the problem. In dynamic algorithms, by contrast, we assume that each process can communicate with some underlying process that is performing some algorithm; and the network's purpose is to carry out some job "servicing" the original algorithm — detecting termination, for example.

14.3.1 Leader Election Algorithms

Our first topic in network algorithms will involve the (static) problem of electing a leader among a set of processes. This is a problem that one might want to solve in any kind of distributed network, but the algorithms that we will explore will stipulate that the processes are distributed in a ring. We also assume that each process begins the algorithm with a unique identifier from some totally ordered set, and that they can communicate with their neighbors in the ring via messages.

A typical example of how a leader-election problem might appear in a real-world setting is in a token ring. A situation might arise in which a token is lost, and has to be regenerated somewhere in the network — but we also want to ensure that only *one* token is regenerated. There are other situations, involving arbitrary network configurations, in which we might want to select some distinguished node in a network; a good example is in spanning tree algorithms, where we might want to designate a "root" process from which we can originate the tree.

For the leader-election problem in a ring, there are a number of varying assumptions that one can make in designing a workable algorithm:

- The number n of processes might be known to each process at the outset of the problem — that is, the number n could be built into each process's local algorithm. Conversely, we might assume that the number of processes in the ring is not known to any processor; the idea here is that the same algorithm should work when the processes are placed in a ring of any size.

- The processes in the algorithm might be capable of bidirectional or only unidirectional (i.e., clockwise or counterclockwise) communication.

- Asynchronous or synchronous processes. (Virtually all the algorithms that we have looked at up until now have assumed asynchronous processes.)

- The identifiers for the processes could be chosen from a bounded set, or instead from (e.g.) the reals or integers.

- The algorithm could be designed to select as leader the process with some particular identifier value (such as the maximum identifier), or it could use some other criterion for selection.

Most of the work in this area has concerned itself with minimizing the number of messages sent in the ring. It is worth pointing out that if the bandwidth in the network is very high, there might be other, more important measures for an algorithm: for instance, total running time.

Note also that the problem of selecting an arbitrary leader among a ring of processes is closely related to the problem of selecting a maximum. Clearly, if a group of processes in a ring can select the one with the maximal identifier, they can designate that process as the leader. Going in the other direction, if a group of processes can select a leader according to some arbitrary criterion, that leader process can send a special "maximum-determining" message around the ring, and thus find the maximum at the cost of n extra messages (beyond those needed to determine the leader).

14.3.2 Le Lann-Chang-Roberts Leader Election Algorithm

The first leader election algorithm that we will discuss is one that we saw toward the end of Lecture 6, due to Le Lann and to Chang and Roberts. In a nutshell, the idea is that each process will send an identifier around the ring. When a node (process) receives an incoming identifier, it compares that identifier to its own; if the incoming identifier is greater than its own, it keeps passing the identifier; but if the incoming identifier is less than its own, it discards the incoming identifier (and does not permit it to continue passing around the ring). A process declares itself the leader when it receives an identifier equal to its own, since this indicates that the identifier has been passed through by every other node in the ring.

Classifying this algorithm according to the dimensions listed above, we see that the Le Lann-Chang-Roberts algorithm:

- Assumes that n, the number of nodes in the ring, is unknown.
- Involves only unidirectional message passing.
- Works asynchronously.
- Uses an unbounded identifier set (e.g., integers).
- Elects the node with the maximal identifier as the leader.

Message-Complexity and Time Analysis for the Le Lann-Chang-Roberts Algorithm

In the worst case, the Le Lann-Chang-Roberts algorithm requires $O(n^2)$ messages to be sent. To see this, suppose that the messages are to be sent clockwise; then the worst-case initial arrangement of identifiers would be that shown in Figure 14.2. In this case, each identifier i is passed approximately i times, so the total number of messages is

$$\sum_{i=1}^{n} i = O(n^2)$$

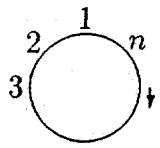


Figure 14.2: The worst case identifier ordering for the Le Lann-Chang-Roberts algorithm.

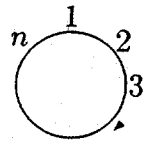


Figure 14.3: The best case identifier ordering for the Le Lann-Chang-Roberts algorithm.

In the best case scenario, the process identifiers are arranged in the opposite order, as in Figure 14.3. In this case, one particular identifier, n, will get all the way around the ring, while every other identifier is blocked by its immediate neighbor. Thus, the total number of messages in this case is only $O(n)$.

The running time for both algorithms, assuming that messages are delivered in parallel around the ring, is $O(n)$. This is just the time taken for the winning identifier to go all the way around the ring.

Since it's pretty clear that there is a wide gap between the best and worst case scenarios for this algorithm (in terms of the number of messages sent), we might be interested in finding the average number of messages sent. The notion of "average performance" here is different than the concept we employed in analyzing the Rabin randomizing algorithms; in that case, we analyzed the performance of an algorithm over all possible *executions*. Now, however, we are interested in averaging the performance of the Le Lann-Chang-Roberts algorithm over all

possible *inputs* — a weaker notion, since instead of an adversary choosing inputs we assume that the inputs are coming in from some random distribution.

To analyze the average performance of the Le Lann-Chang-Roberts algorithm, let's begin by assuming (without loss of generality) that the identifiers are chosen from the set $1, 2, \ldots n$. We further assume that the identifiers are ordered randomly around the ring. The expected total number of messages is just:

$$\sum_{i}^{n} E(i)$$

where $E(i)$ is the expected distance (in links) that identifier i travels before encountering a process whose id-number is greater than i. Clearly, $E(n) = n$, since identifier n will always go all the way around the ring. $E(n-1)$ can be found by noting that identifier $n-1$ will go around the ring until it encounters process n. The expected distance (in links) between process $n-1$ and process n is $n/2$, so $E(n-1) = n/2$.

Continuing, $E(n-2)$ can be found by noting that identifier $n-2$ will travel around the ring until it meets the earlier of process $n-1$ and process n. Intuitively, we'd expect the average distance to the first of these processes to be about $n/3$, and indeed an exact calculation shows that $E(n-2) = n/3$. In general, our intuition suggests that

$$E(i) = \frac{n}{(n-i+1)}$$

Before going on, we can sketch the exact derivation of this expression for $E(i)$. Let j denote $n-i+1$, so that the jth largest id-number is i. Now the problem can be phrased as follows: we have patterns consisting of "dashes" and "X's", where dashes denote nodes with an identifier less than i, and X's denote nodes with an identifier greater than i. We wish to distribute $j-1$ X's in patterns consisting of a total of $n-1$ X's and dashes, and to find the expected position of the first X. A sample pattern, for $n = 10$ and $i = 8$, is shown below:

```
- - - X - - - X -
```

The total number of patterns containing $j-1$ X's is

$$\binom{n-1}{j-1}$$

We wish, therefore, to sum up the total number of messages sent to reach the first X in all patterns, and divide by the total number of patterns to get the average number of messages sent over all patterns.

Now, all patterns cause one message to be sent, so we have

$$\binom{n-1}{j-1}$$

messages contributed by all patterns (that is, there are this many "first messages" sent). The number of patterns that cause a second message to be sent will be just the number of patterns beginning with a dash; so the number of second messages is

$$\binom{n-2}{j-1}$$

Similarly, the number of patterns beginning with *two* dashes will each contribute a "third message" to the total; there are thus

$$\binom{n-3}{j-1}$$

third messages. In general, then, we find that the total number of message for all patterns is:

$$\binom{n-1}{j-1} + \binom{n-2}{j-1} + \ldots \binom{j-1}{j-1} = \binom{n}{j}$$

Thus the average number of messages for a given choice of i is, as predicted by intuition:

$$\frac{\binom{n}{j}}{\binom{n-1}{j-1}} = \frac{n}{j}$$

Therefore the expected total of all messages is the sum of the harmonic series:

$$n + \frac{n}{2} + \frac{n}{3} + \ldots \frac{n}{n} = n(1 + 1/2 + \ldots 1/n) = O(n \log n)$$

14.3.3 Hirshberg-Sinclair Leader Election Algorithm

Having looked at the Le Lann-Chang-Roberts algorithm, a natural question to ask is whether we can do better than $O(n^2)$ as our worst-case message complexity. Is it possible to get a worst-case performance of $O(n \log n)$ messages sent? The first algorithm to show that it was indeed possible (albeit at a sacrifice in terms of running time) was constructed by Hirshberg and Sinclair.

The Hirshberg-Sinclair algorithm can be classified according to the list of leader-election properties that we enumerated before:

- Assumes that n, the number of nodes in the ring, is unknown.
- Involves *bidirectional* message passing.
- Works asynchronously.

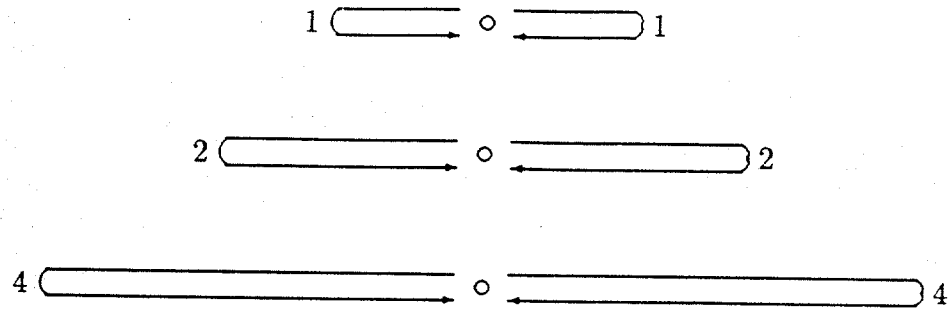


Figure 14.4: Successive message-sends in the Hirshberg-Sinclair algorithm

- Uses an unbounded identifier set.
- Elects the node with the maximal identifier as the leader.

The only difference between the Le Lann-Chang-Roberts and Hirshberg-Sinclair algorithms in this classification is that the latter assumes that we have bidirectional message passing.

Roughly, the idea of the Hirshberg-Sinclair algorithm is that every process, instead of sending messages all the way *around* the ring as in the Le Lann-Chang-Roberts algorithm, will send messages that "turn around" and come back to the originating process. Each process sends out messages (in both directions) that go successively larger distances before returning; in particular, a process first sends messages out for a distance of 1 in both directions; then 2; then 4; and so on, each time doubling the distance of the previous message. This idea is suggested by the sketch in Figure 14.4.

When a message is sent out by a process p, some other process in that message's path may discover that p can't win because its own id-number is greater than that of p. In this case, rather than pass along the original message, it sends back a message to p effectively telling p to stop initiating messages. Similarly, a process q that sees a message with an id-number bigger than its own can deduce that it cannot win, and therefore need not initiate any new messages. Finally, if a process receives its own message (before that message has "turned around"), this means that it is the winner.

It should be clear that this algorithm works, in that it elects as leader only the process with the highest id-number.

Message and Time Analysis for the Hirshberg-Sinclair Algorithm

A process will initiate a message along a path of length 2^i only if it has not been defeated by another process within distance $2^{(i-1)}$ in either direction along the ring. This means that

within any group of $2^{(i-1)} + 1$ consecutive processes along the ring, only one will go on to initiate messages along paths of length 2^i. Thus, at most

$$\left\lceil \frac{n}{2^{i-1}+1} \right\rceil$$

in total will initiate messages along paths of length 2^i.

The total number of messages sent out is then bounded by

$$4\Big((1 * n) + (2 * \lceil \frac{n}{2} \rceil) + (4 * \lceil \frac{n}{3} \rceil) + \ldots (2^i * \lceil \frac{n}{2^{i-1}+1} \rceil) + \ldots\Big)$$

The leading term of 4 in this expression is derived from the fact that each round of message-sending for a given process occurs in both directions — clockwise and counterclockwise — and that each outgoing message must turn around and return. (Thus, for example, in the first round of messages, each process sends out two messages — one in each direction — a distance of one each; and then each outgoing message returns a distance of one, for a net total of four messages sent.) Each term in the large parenthesized expression is the number of messages sent out around the ring at a given pass (counting only messages sent in one direction, and along the outgoing path). Thus, the first term, $(1 * n)$, indicates that all n processes send out messages for an outgoing distance of 1.

Each term in the large parenthesized expression is less than or equal to $2n$, and there are at most $1 + \lceil \log n \rceil$ terms in the expression, so the total number of messages is $O(n \log n)$, with a constant factor of approximately 8.

The time complexity for this algorithm is just $O(n)$, as can be seen by considering the time taken for the eventual winner. The winning process will send out messages that take time 2, 4, 8, and so forth to go out and return; and it will finish after sending out the $\lceil \log n \rceil$th message. If n is an exact power of 2, then the time taken by the winning process is approximately $3n$, and if not the time taken is at most $4n$.

14.3.4 Peterson Leader Election Algorithm

Hirshberg and Sinclair, in their original paper, conjectured that in order to get $O(n \log n)$ worst case performance, a leader election algorithm would have to allow bidirectional message passing. Peterson, however, constructed an algorithm disproving this conjecture. Employing our usual classification scheme, the Peterson algorithm may be summarized as follows:

- Assumes that n, the number of nodes in the ring, is unknown.
- Involves unidirectional message passing.
- Works asynchronously.
- Uses an unbounded identifier set.

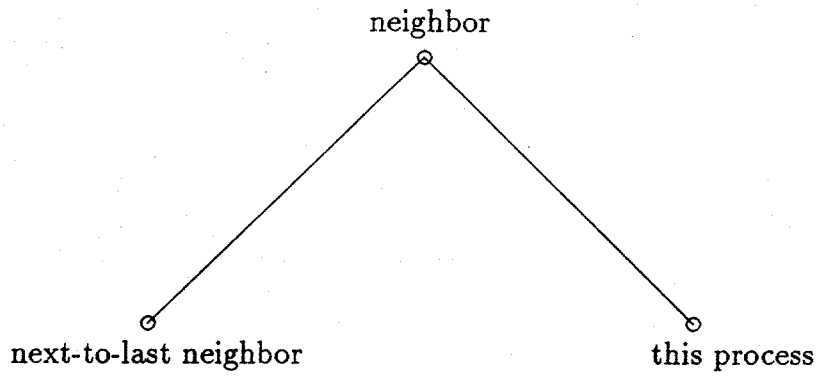


Figure 14.5: A "good" configuration for a Peterson-algorithm process.

- Elects *any* node as leader.

The only difference in this classification scheme between the Le Lann-Chang-Roberts and Peterson algorithms is that the latter may elect as leader any particular node (rather than the one with the maximal id-number, as in the Le Lann-Chang-Roberts algorithm). The Peterson algorithm not only has $O(n \log n)$ worst case performance, but in fact the constant term is low; it is easy to show an upper bound of $2n \log n$, and Peterson used a trickier, optimized construction to get a worst case performance of $1.44n \log n$. (The constant has been brought even further down by other researchers.)

In Peterson's algorithm, processes are designated as being either in an *active* state or *relay* state; all processes are initially active. We can consider the active processes as the ones "doing the real work" of the algorithm, or as the processes still participating in the leader-election process. Relay processes, in contrast, just pass messages along.

The Peterson algorithm is divided into (asynchronously determined) phases. In each phase, the number of active processes will be divided at least in half, so there will be at most $\log n$ phases.

In the first phase of the algorithm, each process sends its id-number two steps clockwise. Thus, everyone can compare its own id-number to that of its two counterclockwise neighbors. When it receives the id-numbers of its two counterclockwise neighbors, each process checks to see whether it is in a configuration such that the immediate counterclockwise neighbor has the highest id-number of the three, as depicted in Figure 14.5. A process in this configuration will remain "active," adopting as a "temporary id-number" the id-number of its immediate counterclockwise neighbor. If not in this configuration, a process becomes a "relay" for the remainder of the execution. The job of a "relay" is to forward messages to active processes.

Subsequent phases proceed in much the same way: among active processors, only those whose immediate (active) counterclockwise neighbor has the highest (temporary) id-number of the three will remain active for the next phase. A process that remains active after a given phase will adopt a new temporary id-number for the subsequent phase; this new id-number will be that of its immediate active counterclockwise neighbor from the just-completed phase.

It is clear that in any given phase, there will be at least one process that finds itself in a configuration allowing it to remain active (unless only one process participates in the phase, in which case the lone remaining process is declared the winner). Moreover, at most half the previously active processes can survive a given phase (since for every process that remains active, there is an immediate counterclockwise active neighbor that must go into its relay state). Thus, as stated above, the number of active processes is at least halved in each phase, until only one active process remains.

A (somewhat abstract) summary of the Peterson algorithm's code is shown below:

```
Active:
temp-id ← initial value;
do forever
    [send(temp-id);
    receive(next-temp-id);
    if next-temp-id = temp-id then announce "leader";
    send(temp-id);
    receive(next-next-temp-id);
    if next-temp-id > max(temp-id, next-next-temp-id)
        then temp-id ← next-temp-id
        else goto relay]

Relay:
do forever
    [receive(temp-id); send(temp-id)]
```

Message and Time Analysis of Peterson's Algorithm

The total number of phases in Peterson's algorithm is at most $\lfloor \log n \rfloor$, and during each phase each process in the ring sends and receives exactly two messages (this applies to both active and relay processes). Thus, there are at most $2n\lfloor \log n \rfloor$ messages sent in the entire algorithm; note that this is a much better constant factor than in the Hirshberg-Sinclair algorithm.

As for time performance, one might first estimate that the algorithm should take $O(n \log n)$ time, since there are $\log n$ phases, and each phase could involve a chain of message deliveries (passing through relays) of net length $O(n)$. As it turns out, however, the algorithm only requires $O(n)$ time.

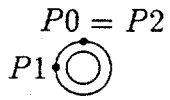


Figure 14.6: The last phase of the Peterson algorithm. $P0$ is the winner.

To do the time analysis of Peterson's algorithm, we begin by assuming an upper bound of 1 on message transmission time; and we assume that internal-processing time is negligible compared to message-transmission time. Now, our plan is to trace backwards the longest sequential chain of message-sends that had to be sent in order to produce a winning process.

Let us denote the eventual winner by $P0$. In the final phase of the algorithm, $P0$ had to hear from two active counterclockwise neighbors, $P1$ and $P2$. In the worst case, the chain of messages sent is actually n in length, and $P2 = P0$, as depicted in Figure 14.6.

Now, consider the previous phase. We wish to continue pursuing the chain backward from $P2$ (which is the same node as $P0$). The key point to recall, though, is that in going from a phase to the previous phase, it must be the case that between any two active processes in the later phase, there is at least one (and possibly two) active processes in the previous phase. Thus, the chain of messages pursued backward from $P2$ in the next-to-last phase can at worst only extend as far as $P1$, as depicted in Figure 14.7. Note also that an additional active process must have existed counterclockwise to $P1$.

At the phase *preceding* the next-to-last phase, there again must have been active processes between $P4$ and $P5$, and between $P5$ and $P2$, as shown if Figure 14.8.

Continuing the chain backwards, we see that each time we move "backward" one phase, we have to insert an active process between every two active processes of the current phase. This means that the chain of messages that eventually terminates with $P0$ in Figure 14.6 can at worst only traverse the entire ring twice. Thus, the total time taken by the algorithm is $O(n)$, with a constant of 2.

14.3.5 An Impossibility Result, and a Lower Bound Result

Having considered some algorithms to solve the leader election problem, we now turn to impossibility and lower bound results for this problem. A well-known result due to Angluin

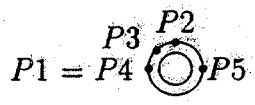


Figure 14.7: The next-to-last phase of the Peterson algorithm.

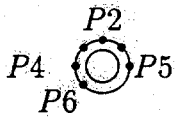


Figure 14.8: The next preceding phase of the Peterson algorithm.

is that it is impossible to elect a leader in a ring in which the processes have no identifiers. The problem is the same one that we encountered earlier in discussing the dining philosophers problem — namely, that in the absence of identifiers it is impossible to break the inherent symmetry of the original ring. This result remains true regardless of whether we assume that the processes know the value of n, or can send bidirectional messages, or operate synchronously, or can conceivably elect any process as leader.

An interesting lower bound result was developed by Burns. In this case, we consider the leader election problem with the following properties:

- Assumes that n, the number of nodes in the ring, is unknown.
- Involves bidirectional message passing.
- Works asynchronously.
- Uses an unbounded identifier set.
- Elects any node as leader.

Burns proved the following:

Theorem 14.3 *A leader election algorithm with the properties listed above must have a worst case performance (in messages sent) of* $(1/4)n \log n$*, where* n *is the number of processes in the ring.*

We assume that n is a power of 2. We will model each process as an I/O automaton, and stipulate that each automaton is distinguishable (in essence, that each process has a unique id-number). The automaton can be represented as in Figure 14.9. Each process has two output messages, **send-right** and **send-left**, and two input messages, **receive-right** and **receive-left**.

Our job will ultimately be to see how a collection of automata of this type behave when connected up into a ring; but in the course of this exploration we would also like to see how the automata behave when arranged *not* in a ring, but simply in a straight line, as in Figure 14.10. Formally, we can say that a line is a linear composition of distinct automata, chosen from the universal set of automata.

We can imagine that the executions of such a line of automata can be examined "in isolation," where the two terminal automata receive no input messages; in this case the line simply operates on its own. Alternatively, we might choose to examine the executions of the line when certain input messages are provided to the two terminal automata.

As an added bit of notation, we will say that two lines of automata are *compatible* when they contain no common automaton between them. We will also define a *join* operation on two compatible lines which simply concatenates the lines; this operation identifies the

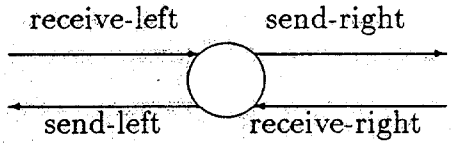


Figure 14.9: A process participating in a leader election algorithm.

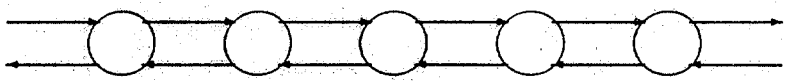


Figure 14.10: A line of leader-electing automata.

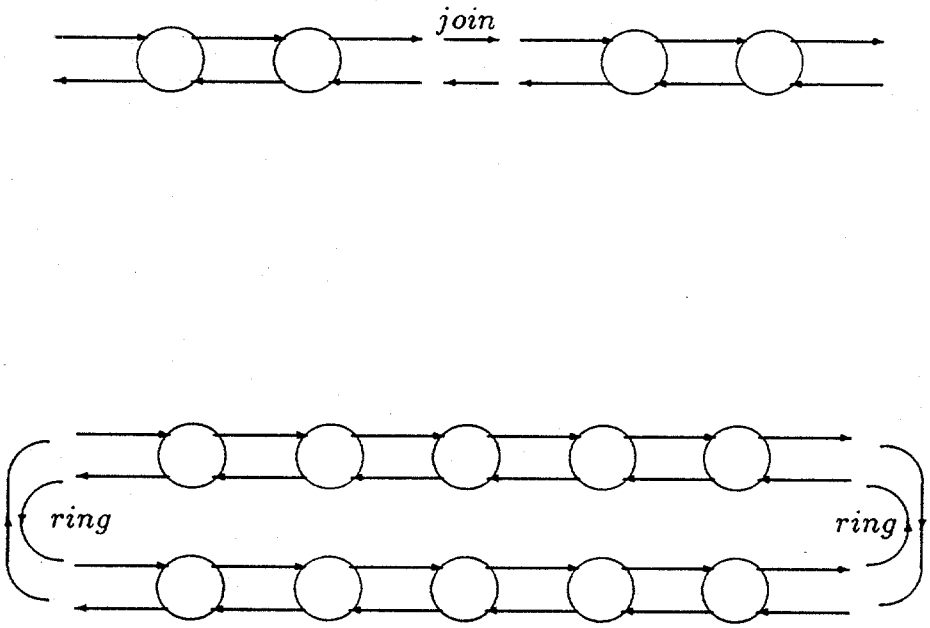


Figure 14.11: Join and ring operations.

rightmost **receive-right** message of the first line with the leftmost **send-left** message of the second, and the leftmost **receive-left** message of the second line with the rightmost **send-right** message of the first. Finally, the *ring* operation on a single line identifies the rightmost **send-right** and leftmost **receive-left** messages of the line, and the rightmost **receive-right** and leftmost **send-left** messages. The *ring* and *join* operations are depicted graphically in Figure 14.11.

The proof of the theorem is given in Lecture 15.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 15: November 3

Lecturers: Nancy Lynch, Hagit Attiya | *Scribe: Magda Nour*

15.1 Burns' Leader Election Message Lower Bound

We proceed with a proof that $\frac{1}{4}n \log n$ messages are required to elect a leader in a bidirectional asynchronous ring, where n is unknown to the processes and process identifiers are unbounded. Recall from Lecture 14, the definitions of a line, a ring, and the join operation. If S is a system (line or ring), and α is an execution of S, then we define:

- MSGS(S) = $\sup_\alpha$ MSGS(S,α).
  Here we consider the number of messages sent during execution. (For lines, we only consider executions in which no messages come in from the ends.)
- A *configuration* q of S consists of the local states and the messages in all buffers.
- A configuration q of a ring is *quiescent* if no execution from q sends any new messages.
- A configuration q of a line is *quiescent* if no execution from q, in which no messages arrive on outside incoming links, sends any new messages.

  Executions from a quiescent configuration can deliver messages already in buffers in S, but generate no new messages. If S is a line, no new messages come in from outside.

Lemma 15.1 *For every $i \geq 0$, there is an infinite set of disjoint lines, $\mathcal{L}_i$, such that for all $L \in \mathcal{L}_i$, $\mid L \mid = 2^i$ and MSGS(L) $\geq 1 + \frac{n}{4} \log n$ (where $n = 2^i$).*

Proof: By induction on i:

Basis: For $i = 0$, we need an infinite set of different processes such that each can send at least 1 message without first receiving one. Suppose, for contradiction, that there are 2 processes, p and q, such that neither can send a message without first receiving one. Consider rings R_1, R_2, and R_3 as shown in Figure 15.1.

In all three rings, no messages are ever sent, so each process proceeds independently. Since R_1 solves election, p must elect itself, and similarly for R_2 and q. Then R_3 elects two leaders, a contradiction. So, at most one process won't send a message before receiving one.

If there is an infinite number of processes, removing one leaves an infinite set of processes that will send a message without first receiving one. Let $\mathcal{L}_0$ be this set, which proves the basis.

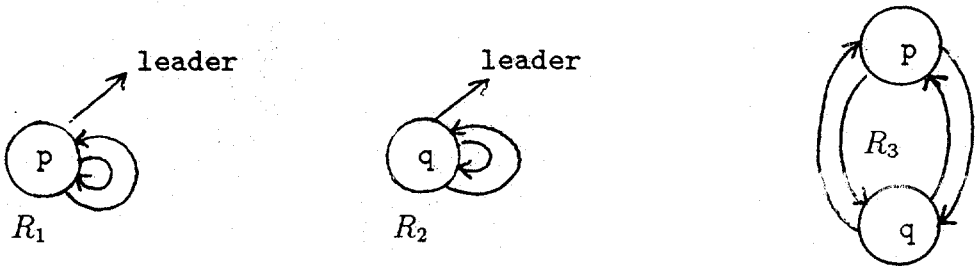


Figure 15.1: Basis for proof of Lemma 15.1

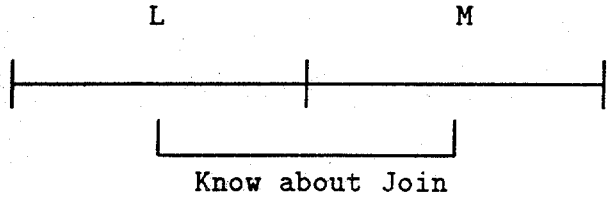


Figure 15.2: Join(L,M)

Inductive step: Assume true for $i-1$. Let $n = 2^i$. Let L, M, N be any 3 lines from $\mathcal{L}_{i-1}$. Consider all possible combinations of two: LM, LN, ML, NL, MN, and NM. Since infinitely many sets of 3 can be chosen from $\mathcal{L}_{i-1}$, the following claim implies the lemma. ■

Claim 15.2 *At least one of the 6 lines can be made to send at least* $1 + \frac{n}{4}\log n$ *messages.*

Proof: Assume false. By the inductive hypothesis, there exists a finite execution α_L of L for which MSGS(L, α_L) $\geq 1 + \frac{n}{8}\log\frac{n}{2}$, and in which no messages arrive from the ends.

We can assume without loss of generality that the final configuration of α_L is quiescent, since otherwise α_L can extend to generate more messages, until $1 + \frac{n}{4}\log n$ messages is exceeded. We can assume the same condition for α_M and α_N by similar reasoning. Now consider any two of the lines, say L and M. Consider join(L,M). Consider an execution that starts by running α_L on L and α_M on M, but delays messages over the boundary.

This gives $\geq 2(1 + \frac{n}{8}\log\frac{n}{2})$ messages. Now deliver the delayed messages. The entire line must quiesce without sending $\frac{n}{4}$ more messages, otherwise the total will be $\geq 2(1+\frac{n}{8}\log\frac{n}{2})+\frac{n}{4}$ $= 2 + \frac{n}{4}\log n$ and the claim is satisfied. This means that at most $\frac{n}{4}$ processes in join(L,M) "know about" the join, and these are contiguous and cross the boundary as shown in Figure 15.2. These processes extend at most halfway into either segment. Let us call this execution α_{LM}. Similarly for α_{LN}, etc.

In ring R_1 of Figure L15:f3, consider an execution in which α_L, α_M, and α_N occur first, quiescing pieces. Then quiesce around boundaries as in α_{LM}, α_{LN}, and α_{NL}. Since the

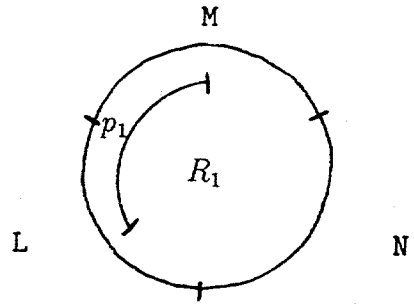


Figure 15.3: Join(L,M,N):Case 1

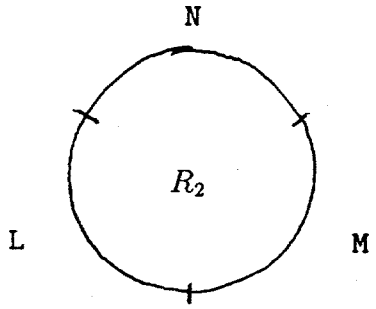


Figure 15.4: Join(L,N,M)

processes that know about each join extend at most half way into either segment, these messages will be noninterfering. Similarly for R_2.

Each of R_1 and R_2 elects a leader, say p_1 and p_2. We can assume without loss of generality that p_1 is between the midpoint of L and the midpoint of M as in Figure 15.3. We consider cases based on the position of p_2 in R_2 (Figure 15.4) to find a contradiction.

- Case 1: p_2 is between the midoint of L and the midpoint of N as in Figure 15.5

  Quiesce as before–run segments and quiesce around boundary.

  No leader is elected in R_3, a ring containing MN. If one is, say p_3, then first suppose it is in lower half as in Figure 15.6. Then it also occurs in R_2 and gets elected there too as in Figure 15.7. There are two leaders in this case which is a contradiction. If it is in the upper half of R_3, then we arrive at a similar contradiction in R_1.

- Case 2: p_2 is between the midpoint of L and the midpoint of M. We arrive at a similar contradiction based on R_3, again.

- Case 3: p_2 is between the midpoint of M and the midpoint of N. We arrive at a similar contradiction based on R_4, a ring containing LN as in Figure 15.8.

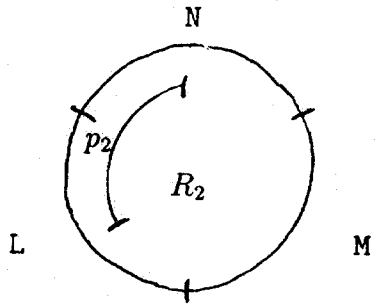


Figure 15.5: Join(L,N,M)

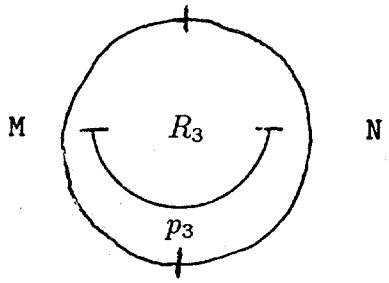


Figure 15.6: Join(M,N): Leader elected in the lower half

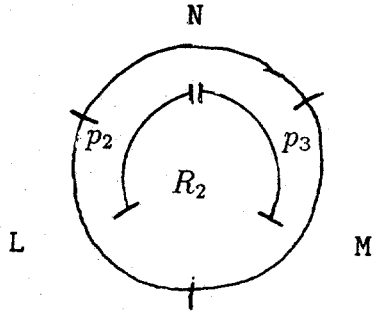


Figure 15.7: Join(L,N,M)

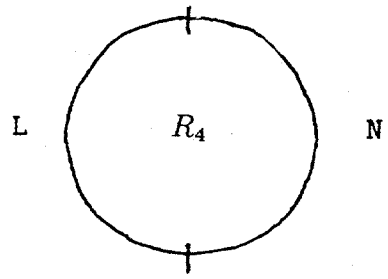


Figure 15.8: Join(L,N): Leader elected in the lower half

■

The reason this suffices is as follows: Let n be a power of 2. Pick a line L of length n with MSGS(L) $\geq 1 + \frac{n}{4} \log n$, and paste it into a circle. Let the processes in L behave exactly as they would if they were not connected, in the execution that sends the large number of messages. Delay all messages across the pasted boundary until all the large number of messages have been sent. Note that his uses asynchrony heavily.

15.2 Synchronous Leader Election Algorithm

15.2.1 Frederickson-Lynch/Vitanyi Algorithm Description

Can Burns' lower bound be applied to a synchronous model? Suppose we have lock-step execution, where processors take steps in a round robin order. Messages are sent round robin, then all processes receive messages round-robin, from the same direction (e.g. first left then right for sending, similar for receiving).

We usually describe synchrony with a model that combines the actions at different locations into one big global action. In other words, at every step, all processes receive all messages from the last step, from left and right, all change state, and all send messages. All of these actions occur atomically. This model gives more power because we can use the fact that a message has *not* arrived to convey some information.

As a first attempt, we assume:

- n is known to all the processes,
- communication is unidirectional,
- process identifiers are integers, and
- all processes start the algorithm at the same time.

The following algorithm can be used: In the first n time slots, only a message marked with identification number 1 can travel. Every process will see a message marked with identification number 1 if one is sent. In general, slots kn, $(k+1)n$ are reserved for messages with identifier k. The smallest identifier is distinguished and therefore the process with that identifier can be elected.

The number of messages is $\leq n$. Unfortunately the time is about nm, where m is the smallest process identifier.

As a second attempt, Frederickson extended these ideas to the case where:

- n is not known,
- communication is unidirectional,
- processes can start algorithm at different times, and
- process identifications are integers.

The number of messages is at most $O(n)$. The time is worse — $O(n2^m)$ where m is the smallest process identification number. This algorithm is not practical, but rather serves to demonstrate limitations on extending the lower bound proof to the synchronous case. We cannot get an $O(n \log n)$ synchronous lower bound without some additional assumptions.

Each process which decides to start participating in the elections spawns a message which will move around the ring, carrying the identification number of the original process.

First idea: Identifiers that originate at different processes are transmitted at different rates. This is implemented by having identifier i travel at the rate of 1 message transmission every 2^i clock pulses.

Any slower identifier that is overtaken by a faster identifier will be deleted (since it has a larger identifier). Also identifier i arriving at process j will be deleted if $j < i$ and process j has also spawned a message process.

Suppose first that all messages are spawned on the same clock pulse. The above strategy then guaranteees that the process with the smallest identifier would get all the way around before the next smaller got half-way, etc, and therefore would use more messages than all the others combined. Therefore total number of messages is $\leq 2n$. Time is $\leq n2^m$.

However, this scheme is not good enough to realize $O(n)$ messages in the case that not all processors spawn their message processes at the same time. If processors with smaller identifiers wake up later than those with larger identifiers, they can still spawn messages that chase and overtake the slower identifiers, but not before about n messages had been expended by each of the participating ($O(n)$) identifiers.

Second idea: Have a preliminary phase before variable rate transmission begins. In the first phase, all identifiers travel at the same rate. Whenever any process decides it wants to participate in the election, it spawns its message and sends it off to its neighbor. The identifier

travels around the ring, until it encounters the next process that had decided to participate. At this point, the identifier continues into the second phase, moving at its variable rate. New entrants after an identifier has already passed by will back off. In other words, a process will not begin to participate after it has received a message from a participating process.

15.2.2 Analysis

Number of Messages

1. The total number of messages in the first phase is n (to close the ring).

2. After no more than n clock pulses from when the first message was spawned, the eventual winner will have entered phase 2.

   Proof: Let p be the first process to spawn and let q be the eventual winner. Since q spawns, it must do so by $d(p,q)$ pulses, where $d(p,q)$ is the distance from p to q. Then q enters the second phase by the time its identifier reaches p (or perhaps a closer process), so there is a total of n messages. ■

3. The total number of second-phase messages sent before the eventual winner enters the second phase is $\leq n$.

   Proof: All processes that are not eventual winners have identifiers ≥ 1. In the n possible clock pulses, identifier i can only travel $\leq \frac{n}{2^i}$ distance. So the total number of messages is $\leq \frac{n}{2^1} + \frac{n}{2^2} + \cdots \leq n$ ■

4. The total number of second-phase messages after the eventual winner enters phase 2 is $\leq 2n$.

   Proof: By relative traveling rates the eventual winner (the process with the smallest identifier) gets all the way around the ring (this takes n messages), the next smaller could send at most $\frac{n}{2}$ messages in that time since its messages are traveling half as fast, the third smaller could send at most $\frac{n}{4}$ messages, etc. The i^{th} smallest processor can send at most $\frac{n}{2^i}$ messages because of its message speed. This is a similar argument as the one for the first algorithm. ■

Therefore, the total number of messages $\leq 4n$.

Time

If m is the identifier of the eventual winner, the algorithm could take $O(n2^m)$ time, which is the same as the first case. Thus, we can achieve $O(n)$ mesages in a synchronous algorithm but with very costly time performance and the use of identifiers for counting.

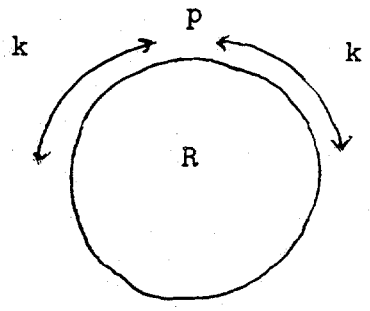


Figure 15.9: A k-neighborhood

15.3 Computing on an Anonymous Ring

Let us consider a slightly different model where processes have no identifiers, but each process has an input $v_i \in \{0,1\}$.

We know that we cannot elect a leader in such a ring but we still can compute functions of the input bits such as OR, XOR, SUM, etc.

A *k-neighborhood* of a process in a ring R is the sequence of input bits of the process and the $2k$ processes surrounding it. Figure 15.9 shows the k-neighborhood of p. In particular, an *0-neighborhood* of a process p_i is v_i.

In the asynchronous model we can control the arrival times of messages. We look at a particular scheduling of message delivery, called the *synchronizing schedule*. This schedule delivers messages in rounds, i.e. all messages sent by the processes based on their initial state are delivered simultaneously. This creates round 1. The messages that result from the receipt of messages at the end of round i are delivered in round $i+1$, for $i \geq 1$.

Lemma 15.3 *If p and q have the same k-neighborhood, then p and q are in the same state after round k.*

Theorem 15.4 *There is no asynchronous algorithm that computes sum correctly on rings of different sizes.*

Proof: Assume there exists some asynchronous algorithm that adds correctly on ring size n_1 and on ring size n_2.

Consider the computation of the algorithm on two configurations of different sizes (n_1 and n_2), with all inputs 1 under the synchronous scheduler. For any k, all k-neighborhoods are identical in both configurations. By the Lemma 15.3, all processes will halt in the same state in both computations and will output the same answer. This is a contradiction.

■

Let us define SYM(R_1, k) to be the smallest number of occurances of any k-neighborhood appearing in R_1.

Two input configurations R_1 and R_2 are a *weak fooling pair* for f (of size n with parameters a and b) if:

1. $f(R_1) \neq f(R_2)$,
2. there exists p_1 in R_1 and p_2 in R_2 with the same $a{\cdot}n$-neighborhood,
3. for all $k, 0 \leq k \leq a{\cdot}n$, SYM($R_1, k$) $\geq b{\cdot}n$. In other words, for all $k, 0 \leq k \leq a{\cdot}n$, any k-neighborhood that appears in R_1 appears at least $b{\cdot}n$ times in R_2.

Theorem 15.5 *If there exists a fooling pair for f, then any asynchronous algorithm for computing f sends at least $\Omega(n^2)$ messages in R_1 in the worst case.*

Proof: Let A be an algorithm for computing f. Look at the computation of A on R_1 and R_2 under the synchronous scheduler. Let T be the first "round" in which no message is sent in R_1. The computation has terminated because there is no message arrival at T hence no state transition at round $T+1$, and at any subsequent round. In particular, all processes in R_1 halt at round T.

If T $\leq a{\cdot}n$, then since p_1 and p_2 have the same $a{\cdot}n$-neighborhood, p_1 and p_2 halt with the same output in both computations. This yields a contradiction. Hence T $\geq a{\cdot}n$.

Let q be the process that sends a message at round k of the computation on R_1. For any round $k \leq T$, some procrss sends a message. Then any process with the same k-neighborhood also sends the same message. By the third fooling pair requirement, there are at least $b{\cdot}n$ such processes. Hence at least $b{\cdot}n$ messages are sent at round k, for $0 \leq k \leq a{\cdot}n$, and so:

$$T = \sum_{k=1}^{a{\cdot}n} b \cdot n = b{\cdot}n \sum_{k=1}^{a{\cdot}n} 1 = b{\cdot}a{\cdot}n^2$$

■

Example: The configurations $R_1 = 1^n$ and $R_2 = 1^{n-1}0$ are a fooling pair for OR (with parameters $b = 1, a = \frac{1}{2}$). From this example we get the following corollary:

Corollary 15.6 *For any n, any algorithm that computes OR on rings of size n sends at least $\frac{n^2}{2}$ messages in the worst case.*

15.4 Exercises

1. Write a bidirectional version of Peterson's leader election algorithm. What are the message and time complexities of your algorithm?

2. Reconsider Burns' lower bound for the number of messages required for electing a leader in an asynchronous ring whose size is a power of 2. What is the best lower bound you can obtain, using the same ideas, for ring sizes that are not powers of two?

3. Assume that the inputs to processors in a synchronous ring are bits. Assume n, the number of processors, is known.

   a. Design and analyze an algorithm to compute the OR of all input bits.
   Your algorithm should use $O(n)$ time and messages.

   b. Design and analyze an algorithm to compute the sum of all input bits; Your algorithm should use $O(n \log n)$ messages.

   Hint: Define "labels" to processors, modify the algorithm from question 1, and use the algorithm of (a) to detect symmetric situations.

4. Consider a "banking system" in which each node of a network keeps a number indicating an amount of money. Messages travel between nodes at arbitrary times, containing money which is being "transferred" from one node to another. Design a distributed algorithm that allows each node to decide on its own balance, in such a way that the total of all the balances is the correct amount of money in the system. Give a convincing argument that your algorithm works. (The algorithm is not allowed to halt or delay transfers.)

*5. Use the Lemma presented in class to give a "higher-level" correctness proof for Bloom's two-writer register algorithm.

6.852 Distributed Algorithms — Fall Semester, 1988

Lecture 16: November 8

Lecturer: Hagit Attiya — *Scribe: Azer Bestavros*

16.1 Computing in Synchronous Rings

In Lecture 15 we considered computation in asynchronous anonymous rings. We proved that for some functions (*e.g.* SUM and XOR) there is no asynchronous algorithms that compute correctly in rings of different sizes. Moreover, we established an $\Omega(n^2)$ lower bound on the number of messages sent by an asynchronous algorithm computing a function f for which a *weak* fooling pair exists, where n – the size of the ring – is known.

It is quite natural to ask whether in the case of synchronous rings there are algorithms that somehow make use of the synchrony to limit the number of messages transmitted. In this section we establish an $\Omega(n \log n)$ lower bound on the number of messages required to compute in synchronous rings.

16.1.1 Active Cycles in Synchronous Algorithms

Let $\mathcal{A}$ be a synchronous algorithm that computes the value of the function f in any configuration. The algorithm $\mathcal{A}$ executes in cycles. In each cycle, the algorithm $\mathcal{A}$ might:

1. Receive messages from its left and/or right neighbors,
2. Perform some local computations based on the information received, and
3. Send messages to its left and/or right neighbors.

Consider the executions of $\mathcal{A}$ on two different configurations R_1 and R_2. Now, assume there is a cycle in the execution of $\mathcal{A}$ where messages are neither sent in R_1 nor in R_2. Obviously, such a cycle does not bring any information that helps $\mathcal{A}$ distinguish between R_1 and R_2.[1] On the other hand, if in a cycle of the computation, messages are sent in R_1 or in R_2 or in both, then such a cycle might carry new information to $\mathcal{A}$ concerning the particular configuration in which it is executing. We define such cycles as *active cycles.*

Lemma 16.1 *Let p and q be processors in $R_1 \cup R_2$, that have the same i-neighborhood then p and q are in the same state after the i^{th} active cycle.*

[1] Note that in some cases an algorithm $\mathcal{A}$ might still acquire information from the fact that no messages are sent – however this information cannot help in differentiating between R_1 and R_2.

Proof: The proof is by induction on the number of active cycles i.

- **Basis:**
  If processors p and q have the same 0-neighborhood then they have the same inputs and thus their initial states are identical.

- **Induction Step:**
  Assume the claim is correct for $i-1$. Thus processors p and q are in the same state after the k^{th} active cycle, where $k \leq (i-1)$. Now consider the i^{th} active cycle. We have 2 cases:

  1. *Neither p nor q got any messages*[2]:
     Obviously, both p and q will remain in the same state or will make the same state transition, since they got no additional information about their neighborhoods.
  2. *P or q got a message*:
     Without loss of generality assume that p got a message from its left (right) neighbor p'. This message was sent at the end of the $(i-1)^{th}$ cycle. Let q' be the left (right) neighbor of q. Since p and q have the same i-neighborhood, it follows that their neighbors p' and q' have the same $(i-1)$-neighborhood. Now, by the induction assumption, both p' and q' should have been in the same state in the $(i-1)^{th}$ cycle and thus should have sent exactly the same messages to their neighbors p and q. Thus, if one of p or q receives a message then the other should also receive the same message. Hence, they both get the same information and thus make the same state transition and remain in the same state.

■

16.1.2 Strong Fooling Pair

Two input configurations R_1 and R_2 are a *strong fooling pair* (of size n and with parameters a and b) for a function f if the following conditions are satisfied:

1. $f(R_1) \neq f(R_2)$.
2. There exists processors p_1 in R_1 and p_2 in R_2 such that p_1 and p_2 have the same an-neighborhood.
3. For all values of k, where $k \leq an$, any k-neighborhood that appears in R_1 or in R_2 appears at least $b\frac{n}{2k+1}$ times in both R_1 and R_2.

[2]Note that this does not interfere with the fact that the i^{th} cycle is active, since it might be the case that some other processors (other that p and q) actually received messages.

Lemma 16.2 *Let $\mathcal{A}$ be a synchronous algorithm computing the function f for which a strong fooling pair (R_1, R_2), with constants a and b, exists. To compute f in a ring of size n, $\mathcal{A}$ will have to execute for at least $a \cdot n$ active cycles.*

Proof: The proof is by contradiction.

- Assume that $\mathcal{A}$ terminates after t cycles, where $t < an$.
- Let p_1 and p_2 be two processors in R_1 and R_2 with the same t-neighborhood. The existence of such processors is guaranteed by property (2) of a fooling pair.
- From Lemma 16.1, both p_1 and p_2 should end up in the same state after the t^{th} active cycle. Thus, they both halt in the same state and should have computed the same value for f. This contradicts property (1) of the fooling pair.

■

Theorem 16.3 *If $\mathcal{A}$ is a synchronous algorithm computing the function f for which a strong fooling pair (R_1, R_2) exists then $\mathcal{A}$ sends at least*

$$\sum_{i=0}^{an} \frac{b \cdot n}{2i+1} = \Omega(n \log n)$$

messages on either R_1 or R_2.

Proof: For any active cycle i, $i \leq an$, let p be a processor that sends a message in this cycle. Such a processor should always exist since i is an active cycle. By property (3) of a fooling pair, there are at least $b \cdot \frac{n}{2i+1}$ other processors with the same i-neighborhood, and by Lemma 16.1 all of these processors will send a similar message. From Lemma 16.2, the number of active cycles is at least $a \cdot n$. Thus, the total number of messages S sent is at least:

$$S \geq \sum_{i=0}^{an} \frac{b \cdot n}{2i+1} \geq \frac{b}{2} \sum_{i=1}^{an} \frac{n}{i}$$

The above summation can be approximated by:

$$S \geq \frac{bn}{2}(\ln an) \geq \Omega(n \log n)$$

■

16.1.3 Lower bound for computing the XOR function

We will show that the computation of the XOR function requires $\Omega(n \log n)$ messages. The idea is to build strong fooling pairs for the XOR function and use Theorem 16.3 to get the $\Omega(n \log n)$ lower bound.

We generate the strong fooling pair using a string substitution homomorphism h which is defined as follows:

$$\begin{aligned} h(0) &\rightarrow 011 \\ h(1) &\rightarrow 100 \end{aligned}$$

Thus,

$$\begin{array}{lcl|lcl} h(0) & = & 011 & h(1) & = & 100 \\ h^2(0) & = & 011100100 & h^2(0) & = & 100011011 \\ h^3(0) & = & 011100100100011011100011011 & h^3(0) & = & 100011011011100100011100100 \\ \ldots & = & \ldots & \ldots & = & \ldots \end{array}$$

Lemma 16.4 *For the string substitution homomorphism h:*

$$\forall r, h^r(0) = \overline{h^r(1)}$$

Proof: The proof is by induction on r. ■

Lemma 16.5 *For the string substitution homomorphism h, any substring σ, $|\sigma| = l$, that appears in either $h^k(0)$ or $h^k(1)$ appears at least $c \cdot \frac{n}{l}$ in both of them.*

Proof:

- First notice that applying the homomorphism h twice on any character (0 or 1) results in a sequence where all strings of length 2 appear cyclically. Now, consider the 3 sequences shown in Figure 16.1, namely: $S_1 = h^k(x)$, $S_2 = h^{k-\log_3 l}(x)$, and $S_3 = h^{k-\log_3 l-2}(x)$, where $x \in 0, 1$.
  - Any string of 2 bits in S_2 contributes at least $2 \cdot 3^{\log_3 l} = 2 \cdot l$ bits in S_1. Thus the substring σ in S_1 results from some 2-bit string in S_2. Let this string be τ.
  - Any single bit (0 or 1) in S_3 contributes at least one τ in S_2.
- From the above analysis (see Figure 16.1) it is obvious that every bit in S_3 contributes at least one σ in S_1. Thus, the number of times σ appears in S_1 is at least equal to the number of bits in S_3 which can be easily found to be:

$$3^{k-\log_3 l-2} = \frac{3^k}{9 \cdot l} = \frac{1}{9} \cdot \frac{n}{l}$$

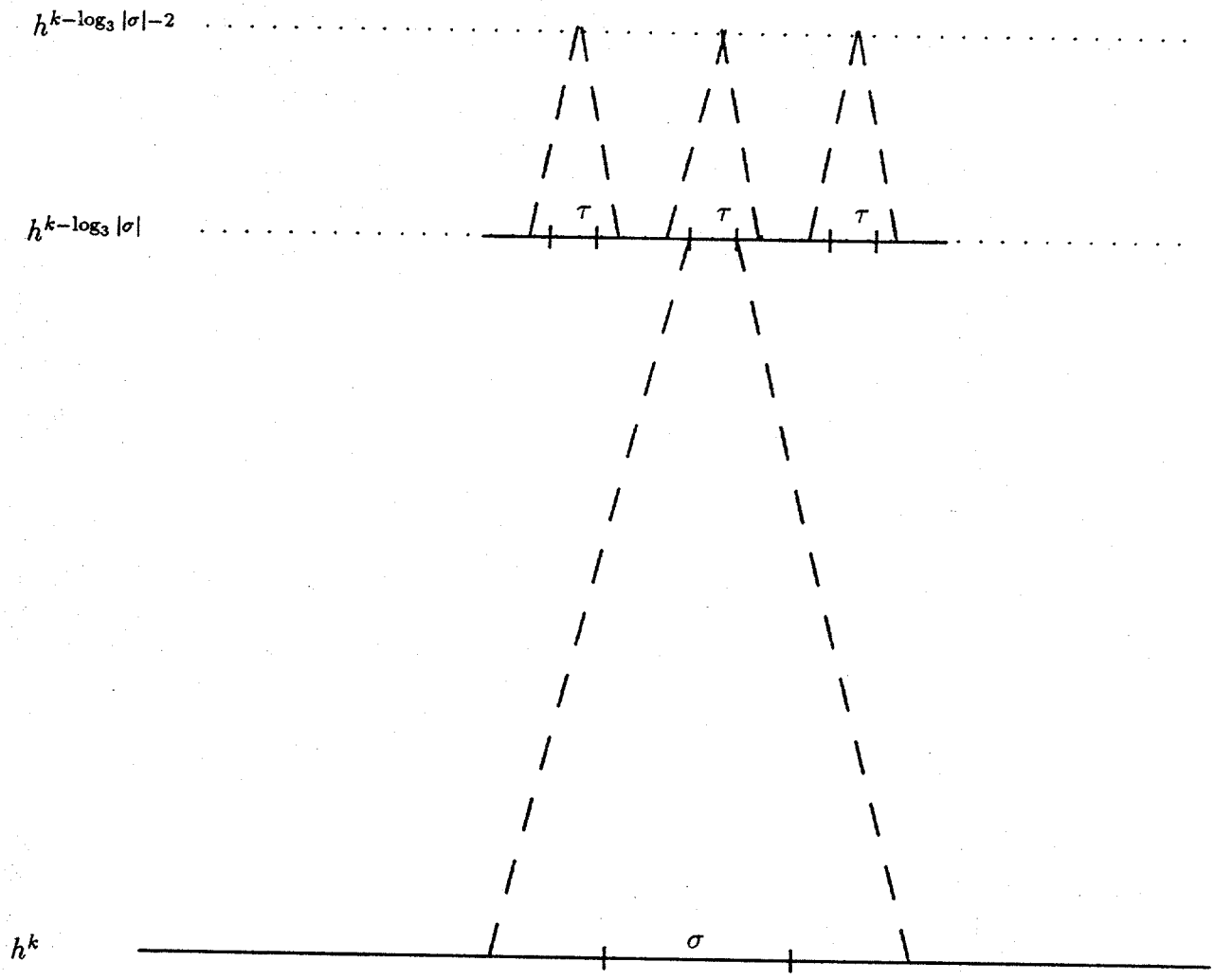


Figure 16.1: Fooling pairs using the string substitution homomorphism h

■

Lemma 16.6 *Let $n = 3^k$ then $(h^k(0), h^k(1))$ is a strong fooling pair for the XOR function.*

Proof: To prove that (R_1, R_2) is a strong fooling pair for the XOR function, we need to show that the three conditions in the definition of strong fooling pairs are satisfied. The first condition, namely $f(R_1) \neq f(R_2)$ follows directly from the fact that $|h^k(0)| = |h^k(1)| = n = 3^k$ is odd and from the fact that $h^k(0) = \overline{h^k(1)}$. The second and third conditions follow directly from Lemma 16.5. ■

Theorem 16.7 *Any synchronous algorithm that computes the XOR function on an anonymous ring of size n sends at least $\Omega(n \log n)$ messages in the worst case.*

Proof: The proof follows directly from Lemma 16.6 and Theorem 16.3. ■

16.1.4 Strong Fooling Pair (revisited)

In our definition of a strong fooling pair, we have assumed that all the processors in the ring are computing the same function f. This is not necessarily true since the algorithm used by the different processors might be computing different values in order to solve a specific problem. For instance, consider the problem of *start synchronization* where processors do not start simultaneously and each has to compute an offset to its local clock so as to bring all clocks to show the same time. Obviously, any algorithm used to solve this problem will result in possibly different values computed by the processors.

The following is a generalized definition for a strong fooling pair that will enable us to talk about solving problems rather than computing functions. Our previous definition is modified so as to make the value computed by the algorithm depend on both the processor executing the algorithm and the configuration in which the algorithm is executed. Thus we define $f(p, R)$ to be the value computed by the algorithm when executed by a processor p in a configuration R.

Definition Two input configurations R_1 and R_2 are a *strong fooling pair* (of size n and with parameters a and b) for a function f if the following conditions are satisfied:

1. There exists processors p_1 in R_1 and p_2 in R_2 such that:
   (a) $f(p_1, R_1) \neq f(p_2, R_2)$, and
   (b) p_1 and p_2 have the same an-neighborhood.
2. For all values of k, where $k \leq an$, any k-neighborhood that appears in R_1 or in R_2 appears $b \cdot \frac{n}{2k+1}$ times in both R_1 and R_2.

16.2 Leader Election in Synchronous Rings

In Lecture 15 we have presented an algorithm by Frederickson and Lynch for leader election in a synchronous ring of size n that uses only $O(n)$ messages.[3] The algorithm uses the processors' identifiers in a non-standard manner and takes a very long time. In this section, we show that these properties are in a sense necessary to achieve the $O(n)$ bound on the number of messages. We will show that if the time is to be bounded or the identifiers are used in a "conservative" manner, then any algorithm for leader election in synchronous rings requires at least $\Omega(n \log n)$ messages in the worst case.

16.2.1 Comparison-Based Algorithms

We assume that every processor in the ring is identical to all the others, except for its own unique *identifier* that is chosen from a totally ordered set L. In any round of the computation, the *state* of a processor records any information known to the processor concerning the identifiers of its neighborhood. Initially, the state of a processor consists of its own identifier. All the processors begin executing the same algorithm at the same time.

In any round of the computation, each processor examines its state and might decide to send messages to its neighbors. Moreover, based on its state and the messages it received from its neighbors (if any), a processor might also change its state. Thus, a synchronous algorithm can be fully described by:

1. A message sending function, and

2. A state transition function.

Definition A synchronous algorithm is called a *comparison-based algorithm* if its message sending and its state transition functions are based solely on comparing the identifiers it has received so far from processors in its neighborhood.

16.2.2 Order Equivalent Neighborhoods

Let X and Y be two neighborhoods of equal size over the totally ordered set L of identifiers. Assume that identical relationships hold between the components of the two neighborhoods. From the above definition, it is obvious that any processor p executing a comparison algorithm, will behave exactly the same in either X or Y. We call such neighborhoods *order*

[3] As opposed to the $\Omega(n \log n)$ lower bound for asynchronous rings.

equivalent.

Definition Two strings $X =< x_1, x_2, \ldots, x_r >$ and $Y =< y_1, y_2, \ldots, y_r >$ are *order equivalent* if and only if:

$$\forall i, j < r, \quad \begin{array}{rcl} x_i < x_j & \iff & y_i < y_j \\ x_i > x_j & \iff & y_i > y_j \end{array}$$

Lemma 16.8 *Let p and q be two different processors executing a comparison-based algorithm $\mathcal{A}$. If p and q have order equivalent k-neighborhoods then p and q have the same state transitions[4] and message generating functions after the k^{th} active cycle.*

Proof: The proof is by induction on k and is very similar to the proof of Lemma 16.1. ∎

16.2.3 Order Fooling Configuration

Definition An input configuration R is an *order fooling configuration* (of size n with parameters a and b) if for any k, $k < an$, if a k-neighborhood appears in R then there are at least $b \cdot \frac{n}{2k+1}$ order equivalent k-neighborhoods that appear in R.

An Example: Consider rings of size n, where $n = 2^l$. Let I be the configuration resulting from assigning to each processor p_i an integer identifier whose binary representation is the reverse binary representation of i. Figure 16.2 shows the identifier assignment for $n = 8$. It is easy to show that I is an order fooling configuration,[5] where $a = \frac{1}{2}$, and $b = \frac{1}{2}$.

Lemma 16.9 *Any synchronous algorithm $\mathcal{A}$ that elects a leader in an order fooling configuration R have at least $a \cdot n$ active cycles.*

Proof: The proof is by contradiction.

- Assume that when $\mathcal{A}$ executes on R, it terminates after t cycles, where $t < an$.
- Let p be a processor that is elected as a leader. Since R is an order fooling configuration, it follows that there exists at least another processor q with order equivalent t-neighborhood. The existence of such a processor is guaranteed by the properties of order fooling configurations.

[4] By the same state transitions we do not mean that they have exactly the same values for their local variables *etc.* Rather we mean that they have equivalent knowledge about the function they are computing.

[5] This is left as an exercise (see problem 8.2).

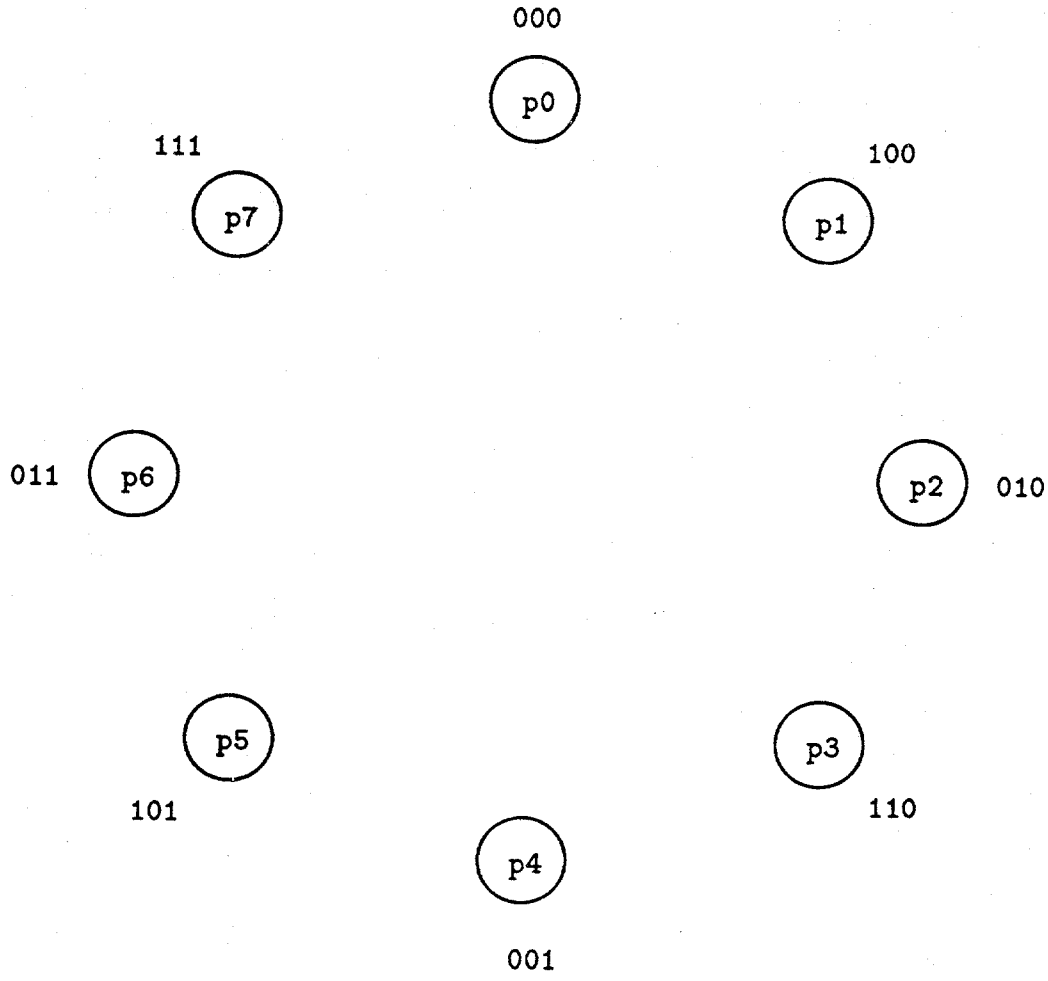


Figure 16.2: Constructing order fooling configuration using the identifier assignment I.

- From Lemma 16.8, both p and q should end up in the same state after the t^{th} active cycle. Thus, they are both elected as leaders – a contradiction.

■

Lemma 16.10 *Any synchronous algorithm $\mathcal{A}$ that elects a leader in an order fooling configuration R of size n sends at least $\Omega(n \log n)$ messages.*

Proof: For any active cycle i, $i \leq a \cdot n$, let p be a processor that sends a message in this cycle. Such a processor should always exist since i is an active cycle. By the properties of order fooling configurations, at least $b \cdot \frac{n}{2i+1}$ other processors have order equivalent neighborhoods, and by Lemma 16.8 all of these processors will send a similar message. From Lemma 16.9, the number of active cycles is at least $a \cdot n$. Thus, the total number of messages S sent is at least:

$$S \geq \sum_{i=0}^{an} \frac{b \cdot n}{2i+1} \geq \frac{b}{2} \sum_{i=1}^{an} \frac{n}{i}$$

The above summation can be approximated by:

$$S \geq \frac{bn}{2}(\ln an) \geq \Omega(n \log n)$$

■

Theorem 16.11 *A lower bound on the number of messages required by an algorithm $\mathcal{A}$ that elect a leader in a synchronous ring of size n is $\Omega(n \log n)$*

Proof: The proof follows directly from Lemma 16.10 if we can show the existence of an order fooling configuration. For the case where n is power of 2, we have already shown how to construct such a configuration – namely using the identifier assignment I. For a general n, it can be shown that an order fooling configuration can be constructed.[6] ■

[6] Refer to the paper by Frederickson and Lynch for a construction.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 17: November 10

Lecturer: Nancy Lynch | *Scribe: Mike Eisenberg*

17.1 Gallager-Humblet-Spira Minimum-Weight Spanning Tree Algorithm

In this lecture, we will examine a distributed algorithm, due to Gallager, Humblet, and Spira [GallagerHS83], for computing the minimum-weight spanning tree (MST) of a graph. The statement of the problem is as follows: let G be an undirected graph with weighted edges in which each vertex is associated with its own processor, and processors are able to communicate with each other via edges. We wish to have the processors (vertices) cooperate to construct a minimum-weight spanning tree for the graph G. That is, we want to construct a subtree covering the vertices in G whose total edge weight is not greater than any other spanning tree for G.

We will assume processes have unique identifiers, and that each edge of the graph is associated with a unique weight known to the vertices on each side of that edge. (The assumption of unique weights on edges is not a strong one given that processors have unique identifiers; for if edges had non-distinct weights, we could derive "virtual weights" for all edges by appending the identifier numbers of the end points onto the edge weights, thereby breaking ties between the original weights.) We will also assume that a process does not know the overall topology of the graph—only the weights of its incident edges—and that it can learn non-local information only by sending messages to other processes over those edges. The output of the algorithm will be a "marked" set of tree edges; every process will mark those edges adjacent to it that are in the final MST.

There is one significant piece of input to this algorithm: namely, that one node will be "awakened" from the outside to begin computing the spanning tree. Nodes do not, therefore, begin computing at the same time—in fact, we assume that the processes work asynchronously. A process can be awakened either by the "outside world" (asking that the process begin the spanning tree computation), or by another, already-awakened process during the course of the algorithm.

There has been a fair amount of work on this problem. The Gallager-Humblet-Spira algorithm focuses on keeping the number of messages sent as small as possible. They achieve a bound of $O((n \log n) + e)$ messages, where n is the number of vertices (processes) and e the number of edges. Intuitively, this is the minimum bound possible: the e term comes from the fact that we have to send a message over each edge in the graph by way of examining

that edge, and the $n \log n$ term comes from the lower bound on the number of messages for leader election that we saw in the Burns theorem proved in Lecture 15. (If we can find a MST in a graph, then we can easily carry out a fan-in procedure to elect a leader. Roughly, the idea here is to have messages sent in "convergecast" fashion inward from the leaves of the tree until they meet at some node which then designated as the root/leader, and which broadcasts its identity back outward along the tree.)

The motivation for this problem comes mainly from the area of communications—the weights of edges might be regarded as "message-sending costs" over the links between processors. In this case, if we want to broadcast a message to every processor, we would use the MST to get the message to every processor in the graph at minimum cost. Additionally, a procedure to solve the MST problem is a useful subroutine for other algorithms—for instance, as mentioned above, it is easy to elect a leader in a graph if that graph happens to be a tree (breaking the symmetry caused by cycles is the hardest part of a general leader-election algorithm). Thus, a reasonable strategy for leader-election in an arbitrary graph might be to first find a spanning tree for the graph, and then find a leader by applying the "convergecast" algorithm described above, sending messages only along the edges of the tree.

The Gallager-Humblet-Spira algorithm is not only interesting, but extremely clever: as presented in their paper, it is about two pages of tight, modular code, and there is a good reason for just about *every* line in the algorithm. In fact, only one or two tiny optimizations have been advanced over the original algorithm. The algorithm has been proven correct via some rather difficult formal proofs (see [WelchLL88]); and it has been referenced and elaborated upon quite often in subsequent research.

17.1.1 A High-Level Description of the Algorithm

We'll begin by examining the Gallager-Humblet-Spira algorithm at a high level of description. Afterward, we'll take a closer look at the code itself.

Two Important Properties of Minimum-Weight Spanning Trees

There is a local property that constitutes the basis for all known (sequential and distributed) minimum-weight spanning tree algorithms:

Property 17.1 *Let G be an undirected graph with vertices V and weighted edges E. Let $(V_i, E_i) : 1 \leq i \leq k$ be any spanning forest for G, with $k > 1$. Fix any i, $1 \leq i \leq k$. Let e be an edge of lowest cost in $E - \bigcup_j E_j$ such that exactly one endpoint of e is in V_i.*

Then there is a spanning tree for G that includes $\{e\} \cup (\bigcup_j E_j)$ and this tree is of as low a cost as any spanning tree for G that includes $\bigcup_j E_j$.

Proof: Suppose the claim were not true—i.e., that T is a spanning tree for G that includes $\bigcup_j E_j$, does not include e and is of lower cost than any other spanning tree including $\bigcup_j E_j$.

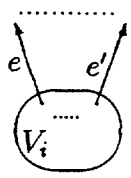


Figure 17.1: Adding edge e to the tree produces a cycle containing e'.

In this case we can add e to our tree T to get a cycle (see Figure 17.1). This cycle contains another edge $e' \neq e$ that is outgoing from V_i.

By hypothesis, we know that $weight(e') \geq weight(e)$. Now, consider T' constructed by deleting e' and adding in e. This is a new spanning tree for the graph that has a weight lower than that of T. Thus, we have arrived at a contradiction: we have a spanning tree including all the edges in the original forest, and that has a weight at least as low as the tree T. ■

To summarize the idea, we can take any spanning forest for the graph G (a collection of disjoint trees that includes every vertex of G); we now choose one of the trees in this collection; and we find the edge of lowest cost with exactly one endpoint in this tree. We'll call this the minimum-weight outgoing edge (MWOE) for this tree. The claim that we have just proved is that there is a spanning tree for G that includes all the edges in the original forest, and that also includes the newly-found edge, *and* that is no larger in cost than any other spanning tree including all the edges in the forest.

This principle forms the basis for well-known sequential MST algorithms. The Prim-Dijkstra algorithm, for instance, starts with one node and successively adds the smallest-weight outgoing edge from the (partially-finished) tree until a complete spanning tree has been obtained. The Kruskal algorithm, by contrast, starts with all nodes as fragments, and successively extends the fragment with the least-weight outgoing edge, thereby combining fragments until there is only one large fragment (the final tree).

Earlier, we noted that in our version of the problem we will assume that all edge weights in our starting graph are distinct. In this case, we have a second property that we can use to simplify our problem:

Property 17.2 *If all edges of a connected graph have distinct weights, then the MST is unique.*

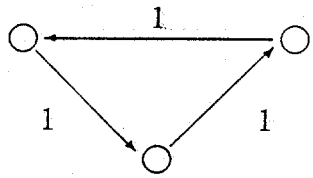


Figure 17.2: An unintended cycle is formed due to edges with equal weights.

Proof: The proof of this property is actually similar to the one above. Suppose there are two trees, T and T', with identical (minimal) weights, and let e be the minimum weight edge found in only one of the two trees. Say (without loss of generality) $e \in T$. Then $e \cup T'$ contains a cycle, and at least one other edge in that cycle, e', is not in T. Since the edge weights are all distinct, and since e' is in one tree but not in the other, we must have $weight(e') > weight(e)$ (by our choice of e). But this implies that $T' \cup \{e\} - \{e'\}$ is a spanning tree with a smaller weight than T', which is a contradiction. ■

Assumptions about the Gallager-Humblet-Spira Algorithm

As noted immediately above, one assumption that we will make for the Gallager-Humblet-Spira MST algorithm is that edge weights are distinct. This property represents a major advantage for parallel MST algorithms: at successive phases, each of a collection of fragments may independently (and simultaneously) choose their own MWOE, combining with other fragments where possible. If the edge weights were not distinct, the fragments couldn't carry out this choice independently, since it would be possible for them to form a cycle unwittingly (as depicted in Figure 17.2).

Besides the use of distinct edge weights, there are some other assumptions used in the Gallager-Humblet-Spira algorithm:

- All nodes operate asynchronously.
- Messages are guaranteed to be delivered eventually, but there is no time bound on delivery.
- Messages are delivered along any particular channel in FIFO fashion (i.e., they are delivered in the order in which they are sent).

- Nodes in the graph receive "wakeup" signals to begin processing; thus, all nodes do not (in general) begin the MST algorithm simultaneously. (This makes the algorithm a little more complicated.)

Connections between the MST Problem and Other Problems

One additional remark about the MST problem is that it has strong connections to two other problems: that of finding *any* spanning tree at all for a graph, and that of electing a leader in a graph.

If you have a spanning tree, it is pretty easy to find a leader; this proceeds via "fan in" of messages from the leaves of the tree until the incoming messages converge on a root node, which can then be designated as the leader. Conversely, if you have a leader, it is easy to find an arbitrary spanning tree: the leader broadcasts messages along each of its neighboring edges, and nodes designate as their parent in the tree that node from which they first receive an incoming message (after which the nodes then broadcast their own messages along their remaining neighboring edges).

A minimum spanning tree is of course a spanning tree; but the converse problem is harder, since an arbitrary spanning tree is not always minimal. How, then, could one find a minimal spanning tree given that one has an arbitrary spanning tree (or a leader)?

One idea would be to have every node send information regarding its surrounding edges to the leader, which then computes the MST centrally and distributes the information back to every other node in the graph. This strategy may seem efficient in terms of the number of messages sent, but realistically it requires a great deal of local computation (on the part of the root node), and the size of the messages sent back from the root node will also be large.

Basic Ideas of the Gallager-Humblet-Spira Algorithm

The central idea of the Gallager-Humblet-Spira algorithm is that nodes form themselves into collections—fragments—of increasing size. (Initially, all nodes are considered to be in singleton fragments.) Each fragment is itself connected by edges that form a MST for the nodes in the fragment. Within any fragment, nodes cooperate in a distributed algorithm to find the MWOE for the entire fragment (that is, the minimum weight edge that leads to a node outside the fragment). The strategy for accomplishing this involves broadcasting over the edges of the fragment, asking each node separately for its own MWOE leading outside the fragment. Once all these edges have been found, the minimal edge among them will be selected as an edge to include in the (eventual) MST.

Once a MWOE for a fragment is found, a message may be sent out over that edge to the fragment on the other side. The two fragments may then combine into a new, larger fragment. The new fragment then finds its own MWOE, and the entire process is repeated until all the nodes in the graph have combined themselves into one giant fragment (whose

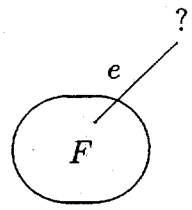


Figure 17.3: How does a node know whether an edge leads outside the fragment?

edges are the MST).

This is not the whole story, of course; there are still some problems to overcome. First, how does a node know which of its edges lead outside its current fragment? A node in fragment F can communicate over an outgoing edge, but the node at the other end needs some way of telling whether it too is in F. (See Figure 17.3.) We will therefore need some way of naming fragments so that two nodes can determine whether they are in the same fragment. But the issue is still more complicated: it may be, for example, that the other node (at the end of the apparently outgoing edge) *is* in F but hasn't learned this fact yet because of communications delays. Thus, some sort of overall synchronization process is needed—some sort of two-phase strategy that ensures that nodes won't search for outgoing edges until all nodes in the fragment have been informed of their current fragment.

Another problem is that the number of messages sent by such an algorithm could be large. The number of messages sent by a fragment to find its MWOE will be proportional to the number of nodes in the fragment. Under certain circumstances, one might imagine the algorithm proceeding by having one large fragment that picks up a single node at a time, each time requiring $\Omega(f)$ messages, where f is the number of nodes in the fragment. (See Figure 17.4.) In such a situation, the algorithm would require $\Omega(n^2)$ messages to be sent overall.

This second problem should suggest a "balanced-tree algorithm" solution: that is, the difficulty derives from the merging of data structures that are very unequal in size. The strategy that we will use, therefore, is to merge fragments of roughly equal size. Intuitively, if we can keep merging fragments at nearly equal size, we can keep the number of total messages to $O(n \log n)$.

The trick we will use to keep the fragments at similar sizes is to associate *level numbers* with each fragment. We will say that if $level(F) = l$ for a given fragment F, then the number of nodes in F is greater than or equal to 2^l. Initially, all fragments are just singleton nodes at

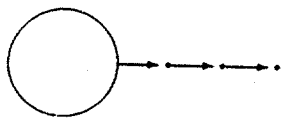


Figure 17.4: How do we avoid a big fragment growing by one node at a time?

level 0. When two fragments at level l are merged together, you get a new fragment at level $l+1$. (This preserves the condition that we specified for level numbers: if two fragments of size at least 2^l are merged, you get a new fragment of size at least 2^{l+1}.)

These level numbers, as it turns out, will not only be useful in keeping things balanced, but they will also provide some identifier-like information helping to tell nodes whether they are in the same fragment.

Before describing the specific messages used in the Gallager-Humblet-Spira algorithm, we can take an overall look at how fragments are combined together. There are two ways of combining fragments:

1. *Merging.* This is the "standard" way of combining. In this case we have two fragments F and F', and they find that they share the same minimum-weight outgoing edge:

$$level(F) = level(F') = l$$

$$MWOE(F) = MWOE(F')$$

   Then it is okay to combine the two fragments into a new fragment at a level of $l+1$.

2. *Absorbing.* There is another case to consider. It might be that some nodes are forming into huge fragments via merging, but isolated nodes (or small fragments) are lagging behind at a low level. In this case, the small fragments may be absorbed into the larger ones without determining the MWOE of the large fragment.

   The rule for absorbing is that if you have two fragments F and F', with $level(F) < level(F')$, and the MWOE of F leads to F', then you can absorb F into F' by combining them along the MWOE of F. The larger fragment formed is still at the level of F'. In a sense, we don't want to think of this as a "new" fragment, but rather just an augmented version of F'.

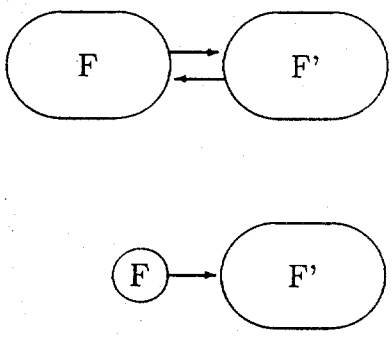


Figure 17.5: Two fragments combine by merging; a fragment absorbs itself into another

These two combining strategies are illustrated (in a rough way) by Figure 17.5. It is worth underlining the fact that just because $level(F) < level(F')$, we do not know that fragment F is smaller than F'; in fact, it could be larger. (Thus, the depiction of F as a "small" fragment in Figure 17.5 is meant only to suggest the typical case.)

If a fragment finds that its MWOE leads to a fragment at a smaller level than itself, it simply holds up and takes no action; thus, the only way in which fragments combine is via merging (in which two fragments of equal level combine) and absorbing (in which a "small" fragment adds itself onto a "large" one).

Level numbers thus serve, as mentioned above, as identifying information for fragments. For fragments of level 1 or greater, however, the specific fragment identifier is the *core edge* of the fragment. The core edge is just the edge along which the merge operation resulting in the current fragment level took place. (Since level numbers for fragments are only incremented by merge operations, we know that any fragment of level 1 or greater must have had its level number specified by some previous merge along an edge; this is the core edge of the fragment.) The core edge also serves as the site where the processing for the fragment originates and where information from the nodes of the fragment is collected.

To summarize the way in which core edges are identified for fragments:

- For a *merge* operation, *core* is the common MWOE of the two combining fragments.
- For an *absorb* operation, *core* is the core edge of the fragment with the larger level number.

We now want to show that this strategy, of having fragments merge together and absorb

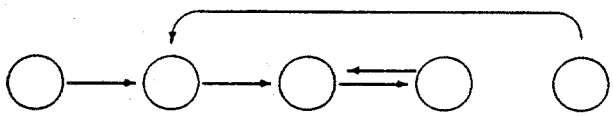


Figure 17.6: A collection of fragments and their minimum-weight outgoing edges.

themselves into larger fragments, will in fact suffice to combine all fragments into a MST for the entire graph.

Claim 17.3 *If we start from an initial situation in which each fragment consists of a single node, and we apply any possible sequence of merge and absorb steps, then there is always some applicable step to take until the result is a single fragment containing all the nodes.*

Proof: We want to show that no matter what configuration we arrive at in the course of the algorithm, there is always some merge or absorb step that can be taken.

One way to see that this is true is to look at all the current fragments at some stage in the running algorithm. Each of these fragments will identify its MWOE leading to some other fragment. If we view the fragments as vertices in a "fragment-graph," and draw the MWOE for each fragment, we get a directed graph with an equal number of vertices and edges. (See Figure 17.6) By the pigeonhole principle, such a directed graph *must* have a cycle; and because the edges have distinct weights, only cycles of size 2 (i.e., cycles involving two fragments) may exist. Such a 2-cycle represents two fragments that share a single MWOE.

Now, it must be the case that two fragments in any 2-cycle can be combined. If the two fragments in the cycle have the same level number, a merge operation can take place; otherwise, the fragment with the smaller level number can absorb itself into the fragment with the larger one. ■

Let's return to the question of how the MWOE is found for a given fragment. The basic strategy is this: each node in the fragment is going to find its own MWOE leading outside the fragment; then we will collect the information from each node at a selected processor, and take the minimum of all the edges suggested by the individual nodes.

This sounds straightforward, but it reopens the question of how a node knows that a given edge is outgoing—that is, that the node at the other end of the edge lies outside the

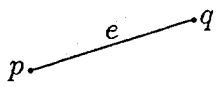


Figure 17.7: Node p wants to know if q is in the same fragment.

current fragment. Suppose we have a node p that "looks across" an edge e to a node q at the other end. (See Figure 17.7.) How can p know if q is in a different fragment or not?

A fragment name (or identifier) may be thought of as a pair $(core, level)$. If q's fragment name is the same as p's, then p certainly knows that q is in the same fragment as itself. However, if q's fragment name is different from that of p, then it is still possible that q and p are indeed in the same fragment, but that q has not yet been informed of that fact. That is to say, q's information regarding its own current fragment may be out of date.

However, there is an important fact to note: if q's fragment name has a core unequal to that of p, and it has a level value at least as high as p, then q can't be in the fragment that p is in currently, and never will be. This is so because, in the course of the algorithm, a node will only be in one fragment at any particular level. Thus, we have a general rule that q can use in telling p whether both are in the same fragment: if the value of $(core, level)$ for q is the same as that of p then they are in the same fragment, and if the value for *core* is different for q and the value of *level* is at least as large as that of p then they are in different fragments.

The upshot of this is that $MWOE(p)$ can be determined only if $level(q) \geq level(p)$. If q has a lower level than p, it simply delays answering p until its own level is at least as great as p's.

The fact that q may delay answering p means that we have to reconsider our earlier argument that the algorithm must make progress until a MST is found. Since a fragment can be delayed in finding its MWOE (since some individual nodes within the fragment are being delayed), we might ask whether it is possible for the algorithm to reach a state in which a merge or absorb operation is not possible. To see that this is not the case, though, we can use essentially the same argument as before, but this time we need only consider those MWOE's found by fragments with the lowest level in the graph (call this level LO). If a fragment at level LO finds a MWOE to a higher-level fragment, then an absorb operation

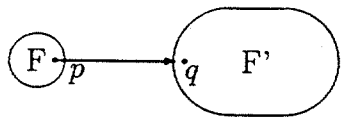


Figure 17.8: Fragment F absorbs itself into F' while F' is still searching for its own MWOE.

is possible; and if all of the fragments at the level LO have a MWOE to some other fragment at level LO, then again we must have a 2-cycle between two fragments at level LO, and a merge operation is possible. So again, we conclude that the algorithm must make progress until the complete MST is found.

Getting back to the algorithm itself, each fragment F will find its overall MWOE by taking a minimum of the MWOE for each node in the fragment. This will be done by a "broadcast-convergecast" algorithm starting from the core, emanating outward, and then collecting all information back at the core.

This leads to yet another question: what happens if a "small" fragment F gets absorbed into a larger one F' while F' is still in the course of looking for its own MWOE? (See Figure 17.8.)

There are two cases to consider (consult Figure 17.8 for the labelling of nodes). Suppose first that $MWOE(q)$, the minimum edge leading outside the fragment F', has not yet been determined. In this case, we must search for a MWOE for the fragment F' in F as well. Since q doesn't yet know which is its own local MWOE, there is still a possibility that e is q's MWOE, and thus the MWOE for the entire fragment F' might emanate from one of the newly-incorporated nodes in F.

On the other hand, suppose $MWOE(q)$ has already been found at the time that F absorbs itself into F'. In that event, the MWOE for q cannot possibly be e, since the only way that the MWOE for q could even be known is for that edge to lead to a fragment with a level at least as great as F'; and we know that the level of F is smaller than that of F'. Moreover, the fact that the MWOE for q is not e implies that the MWOE for the entire fragment F' cannot possibly be in F. This is true because e is the MWOE for fragment F, and thus there can be no edges leading out of F with a smaller cost than the already-discovered MWOE for node q. Thus, we conclude that if $MWOE(q)$ is already known at the time the absorb operation takes place, then fragment F' needn't look for its overall MWOE

among the newly-absorbed nodes. This is fortunate, since if F' did in fact have to look for its MWOE among the new nodes, it could easily be too late: by the time the absorb operation takes place, q might have already reported its own MWOE, and fragment F' might already be deciding on an overall MWOE without knowing about the newly-absorbed nodes. However, since F' does not in fact have to worry about these new nodes in this case, the algorithm continues to work correctly.

A Summary of the Code in the Gallager-Humblet-Spira Algorithm

We have seen the major intuitive ideas of the Gallager-Humblet-Spira algorithm, and the presentation above should be sufficient to guide the reader through the code presented in their original paper. Although the actual code in the paper is dense and complicated, the possibility of an understandable high-level description turns out to be fairly typical for communications algorithms. In fact, the high-level description that we have seen can serve as a basis for a correctness proof for the algorithm. (Attempting a correctness proof based directly on the low-level code itself would be a good deal more difficult.)

The following message types are employed in the actual code:

- INITIATE messages are broadcast outward on the edges of a fragment to tell nodes to start finding their MWOE.
- REPORT messages are the messages that send the MWOE information back in (these represent the convergecast response to the INITIATE broadcast messages).
- TEST messages are sent out by nodes when they search for their own MWOE.
- ACCEPT and REJECT messages are sent in response to TEST messages from nodes; they inform the testing node whether the responding node is in a different fragment (ACCEPT) or is in the same fragment (REJECT).
- CHANGE-ROOT is a message sent toward a fragment's MWOE once that edge is found. The purpose of this message is to change the root of the (merging or currently-being-absorbed) fragment to the appropriate new root.
- CONNECT messages are sent across an edge when a fragment combines with another. In the case of a merge operation, CONNECT messages are sent both ways along the edge between the merging fragments; in the case of an absorb operation, a CONNECT message is sent by the "smaller" fragment along its MWOE toward the "larger" fragment.

In a bit more detail, INITIATE messages emanate outward from the designated "core edge" to all the nodes of the fragment; these INITIATE messages not only signal the nodes

to look for their own MWOE (if that edge has not yet been found), but they also carry information about the fragment identity (the core edge and level number of the fragment). As for the TEST-ACCEPT-REJECT protocol: there's a little bookkeeping that nodes have to do. Every node, in order to avoid sending out redundant messages testing and retesting edges, keeps a list of its incident edges in the order of weights. The nodes classify these incident edges in one of three categories:

- *Branch* edges are those edges designated as part of the building spanning tree.
- *Basic* edges are those edges that the node doesn't know anything about yet — they may yet end up in the spanning tree. (Initially, of course, all the node's edges are classified as basic.)
- *Rejected* edges are edges that cannot be in the spanning tree (i.e., they lead to another node within the same fragment).

A fragment node searching for its MWOE need only send messages along basic edges. The node tries each basic edge in order, lowest weight to highest. The protocol that the node follows is to send a TEST message with the fragment level-number and core-edge (represented by the unique weight of the core edge). The recipient of the TEST message then checks if its own identity is the same as the TESTer; if so, it sends back a REJECT message. If the recipient's identity (core edge) is different and its level is greater than or equal to that of the TESTer, it sends back an ACCEPT message. Finally, if the recipient has a different identity from the TESTer but has a lower level number, it delays responding until such time as it can send back a definite REJECT or ACCEPT.

When two CONNECT messages cross, this is the signal that a merge operation is taking place. In this event, a new INITIATE broadcast emanates from the new core edge and the newly-formed fragment begins once more to look for its overall MWOE. If an absorbing CONNECT occurs, from a lower-level to a higher-level fragment, then the node in the high-level fragment knows whether it has found its own MWOE and thus whether to send back an INITIATE message to be broadcast in the lower-level fragment.

Message Complexity of the Gallager-Humblet-Spira Algorithm

In order to analyze the message complexity of the Gallager-Humblet-Spira algorithm, we have to apportion the messages into two different sets, resulting separately (as we will see) in the $O(n \log n)$ term and the $O(e)$ term.

The $O(e)$ term arises from the fact that each edge in the graph must be tested at least once: in particular, we know that TEST messages and associated REJECT messages can occur at most once for each edge. Thus we get an $O(e)$ term resulting from the 2 messages (the TEST-REJECT pair) over each edge. (It is important to recall in this regard that once a REJECT message has been sent over an edge, that edge will never be tested again.)

All other messages sent in the course of the algorithm—the TEST-ACCEPT pairs that go with the acceptances of edges, the INITIATE-REPORT broadcast-convergecast messages, and the CHANGEROOT-CONNECT messages that occur when fragments combine—can be considered as part of the overall process of finding the MWOE for a given fragment. In performing this task for a fragment, there will be at most one of these messages associated with each node (each node receives at most one INITIATE and one ACCEPT; each sends at most one successful TEST, one REPORT, and one of either CHANGEROOT or CONNECT). Thus, the number of messages sent within a fragment in finding the MWOE is $O(f)$ where f is the number of nodes in the fragment.

The total number of messages sent in the MWOE-finding process, therefore, is

$$\sum_{\text{all fragments } F} \text{number of nodes in } F$$

which is

$$\sum_{\text{all level numbers } L} \left(\sum_{\text{all fragments } F \text{ of level } L} \text{number of nodes in } F \right)$$

Now, the total number of nodes in the inner sum at each level is at most n, since each node appears in at most one fragment at a given level-number L. And since the biggest possible value of L is $\log n$, the sum above is bounded by:

$$\sum_{1}^{\log n} n = O(n \log n)$$

Thus, the overall message complexity of the algorithm is $O(e + (n \log n))$.

Proving Correctness for the Gallager-Humblet-Spira Algorithm

A good deal of interesting work remains to be done in the field of proving correctness for communications algorithms like the Gallager-Humblet-Spira algorithm. One promising approach—and an approach that, in fact, works for this particular algorithm—is to prove correctness for a high-level description of the algorithm (e.g., at the graph level) using invariant-assertion and other standard techniques. Having done that, one can prove independently that the code in fact correctly simulates the high-level description. (See [WelchLL88].)

This latter stage of proof can be formalized by implementing the high-level algorithm within an I/O automaton (in which the state consists of fragments, and actions include merge and absorb operations); implementing the low-level code in another I/O automaton; and then showing that there is a mapping from sets of actions of the low-level automaton to the high-level one such that fair behaviors of the low-level automaton correspond to fair behaviors of the high-level automaton.

17.2 Dynamic Network Algorithms: Distributed Snapshots

So far we've been talking about static network algorithms. Now we're going to look at algorithms with a more dynamic nature in that they are designed to interact with some other, ongoing distributed algorithm.

The first algorithm we will consider is one for computing distributed snapshots, due to Chandy and Lamport. The idea is that we want to determine a "consistent global state" for a distributed system running some algorithm. Our model for the system of interest is a set of processes communicating via messages over FIFO channels; we could think of these as I/O automata with SEND and RECEIVE actions.

An important question, of course, is defining just what we mean by a "consistent global state"—in particular, what is a "global state," and what do we mean by "consistency"?

We think of a *global state* as a state for all nodes and all channels in the system—i.e., the values of variables in the nodes of the system, and the particular messages being sent along the channels of the system.

The notion of *consistency* is a bit subtler; to define this term, we have to go back to Lamport's earlier notion of *logical time*. Recall that a logical time ordering for events in a system is a total ordering for the events with the following properties:

- Events at any particular node are ordered in order of occurrence at that node.
- SEND actions for a particular message are ordered before RECEIVE actions for that message.
- Only finitely many events can occur before any particular event.

This definition implies that the same execution could have many possible logical time orderings assigned to it.

Each consistent state is going to arise from a particular logical time assignment at a particular time. That is, we look at some execution; we find some way of assigning a logical time to all the events; and then we pick a particular time, and freeze what's happening in the system at that logical time. Note that this may not correspond to any *real* time—it may not correspond to the actual order in which the events occurred—but there is some way that you *could* have assigned the times to the events in the execution that satisfies the logical time properties. The information that you get for a consistent state at some time, then, is the information for a logical time assignment at that particular time.

So, once you fix a logical time assignment and a particular time, the consistent state that you want is the states of the nodes up through time t; and the channel state consists of those messages that were sent but not yet received at time t, in order of sending.

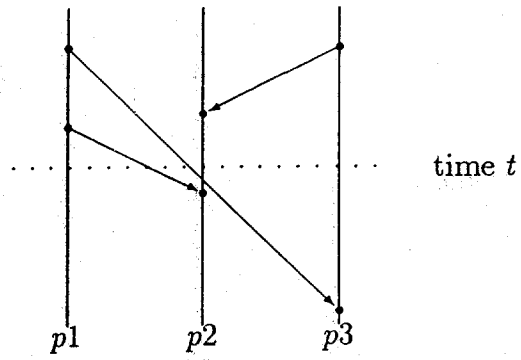


Figure 17.9: An execution represented by individual process timelines.

One way to think of the snapshot notion is to imagine a picture of the execution drawn as a set of timelines, one for each process (see Figure 17.9). The events that happen at each process are consistent with those in the given asynchronous execution. We can now imagine that the timelines are stretched and shrunk in various places individually so that logical time t corresponds to a horizontal line: all events after time t appear below the line, and all events before time t are above the line. Again, even though the events may not really have occurred in the order depicted, as far as each node is concerned the diagram is consistent with the events seen at that node.

We want more than just a consistent global state: after all, the initial state of the system fits this definition. What we want is, loosely, a *recent* consistent global state—a state that conveys information about the system at some recent time. For instance, we might like the result of our snapshot algorithm to reflect all the events that occurred in real time before the algorithm began running.

Now, why would we want to get a global snapshot of a system? One application might occur in maintaining databases: we might like to get a consistent state of a distributed database (e.g., a bank audit over multiple branches of a bank). Another use is in deadlock detection: you can use a global snapshot to find out if every node is (or might conceivably be, in some execution) waiting for a result from some other node in order to proceed. Yet another idea is to use a global snapshot algorithm to detect the termination of some distributed algorithm: the snapshot might show, for instance, that each node is in an idle state and no messages are in transit, in which case we know that the algorithm has terminated. Trying to determine this by querying the nodes individually is problematic: a node might tell us that it is currently idle, but it might receive an incoming message as soon as we have moved on to query another node.

17.3 Exercises

1. Look at the synchronous anonymous ring (n known). Assume processors do not start simultaneously, and consider the problem of *start synchronization:*

   > Each processor has to computes an offset to its local clock, so as to bring all clocks to show the same time.

   Show that any algorithm that achieves start synchronization sends at least $\Omega(n \log n)$ messages in the worst case.

2. Consider comparison algorithms for leader election on synchronous rings of size $n = 2^l$ (where processors have distinct identifiers).

   (a) Show the following for the identifier assignment, I, that was described in class: For any $k < \frac{n}{2}$, if a k-neighborhood appears in I, then there are at least $b \cdot \frac{n}{k}$ order equivalent k-neighborhoods that appear in I (calculate the constant b).

   (b) Show that if the k-neighborhoods of processors p and q (in the same configuration) are order-equivalent, then p and q are in the same state after k active cycles.

   (c) Combine (a) and (b) to show that any such algorithm must send at least $\Omega(n \log n)$ messages on I.

3. Consider the problem of electing a leader in a synchronous ring of size n, where n is known to all the processes and the processes have no id's. Devise a randomized leader election algorithm that has the best expected time and message performance that you can achieve.

4. Consider the Gallager et al minimum spanning tree algorithm.

   (a) State and prove an upper bound on the time from when the first node wakes up until the last node announces its results. (Assure upper bounds of 1 on message delivery time and 0 on local processing time.)

   (b) How tight is your upper bound proved in (a)?
   That is, describe a particular execution of the algorithm in which the best upper bound that can be proved on time, (subject to the assumptions above) is as large as you can find.

*5. In the Gallager algorithm, why is the conditional test needed in the line of code "if SE(j)=Basic then SE(j) $\leftarrow$ Rejected " which appears in the Test and Reject subroutines? That is, why doesn't that line read simply " SE(j) $\leftarrow$ Rejected " ?

6.852 Distributed Algorithms	Fall Semester, 1988
Lecture 18: November 15	
Lecturer: Nancy Lynch	*Scribe: Jeff Fried*

18.1 Global Snapshots

At the end of Lecture 17, we introduced the idea of a *global snapshot*, or a consistent global state of a distributed system. In many applications, such as databases, termination detection, and deadlock detection, it is crucial to be able to capture a state which is both *consistent* and *recent* (these terms will be more formally defined soon). Algorithms and conditions for global snapshots have been explored in [ChandyL85,FischerGL82]; in these approaches, each processing node in a system snapshots its own state and the state of its incoming communication channels. The trick is to find some rules defining when and how to have each node capture these states.

The model we assume is communicating processes with messages sent over FIFO links with guaranteed message delivery. A *global state* consists of states for all nodes and all channels in the system.

18.1.1 What could go wrong in capturing a global snapshot

In order to understand how to construct a global snapshot, let's consider what could go wrong with the naive approach via a few examples. Consider an audit algorithm running on two nodes, p and q, with two communication channels, C and D. Assume we start with a single dollar at p, as shown in Figure 18.1 (this is a poor rural bank). Consider the following sequence:

1. p records its own state (as \$1)
2. p sends the dollar to q on channel C
3. q receives the dollar over channel C
4. q records its own state (as \$1)

When the local state of the processes is collected, it looks as though there are two dollars in the system! This is bad because it is not a *consistent* state.

What does it mean for a global state to be *consistent*? Loosely, each consistent state consists of the state of the processors and channels at a particular time from a particular logical time assignment.

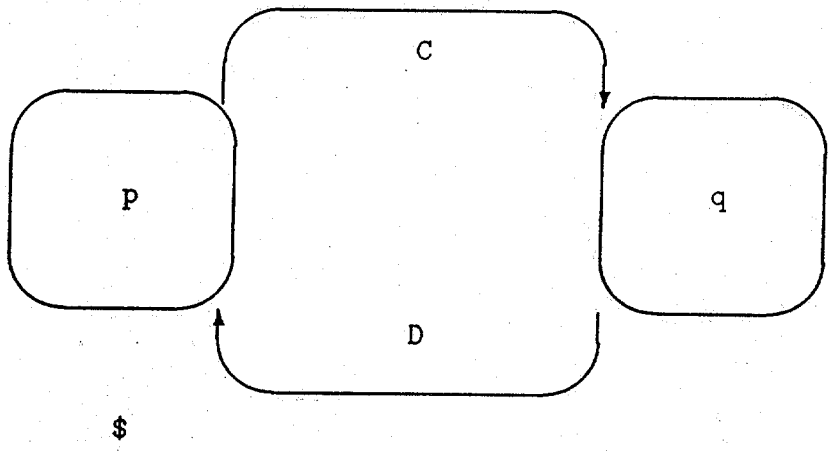


Figure 18.1: Example of global audit (initial conditions)

Definition A *logical time assignment* is a global time ordering of events which has the following properties:

1. Events at every node are ordered in their local order of occurrence.
2. Every send action is ordered before its corresponding receive action.
3. Only finitely many many events are ordered before any particular event.

A consistent global state will correspond to a time for some logical time assignment. In the example with banks p and q, the problem arises because messages sent between the processes are counted inconsistently (the same dollar is counted more than once). The snapshot collected in this case did not correspond to any time in any logical time assignment.

Let $n \equiv$ the number of messages sent on C before p's state is recorded

Let $n' \equiv$ the number of messages sent on C before q's state is recorded

Then $n < n'$ causes the above problem. We need any global snapshot algorithm to ensure that $n' \leq n$. We could also have a similar problem if $n' < n$:

1. q records its own state (as \$0)
2. p sends the dollar to q on channel C
3. p records its own state (as \$0)

This time, it looks as though the bank is broke; to ensure that messages are not lost in the snapshot, we need $n' = n$.

Since every process records the state of its incoming communication channels as well as its own state, we need to insure that messages are counted consistently between processes and their incoming channels.

Let $m \equiv$ the number of messages received by q along C before q's state is recorded

Let $m' \equiv$ the number of messages received by q along C before C's state is recorded

To ensure that the state of processes and their incoming channels is consistently recorded, we need $m' = m$. Because we cannot receive more messages than were sent along a channel, we have $n = n' \geq m' = m$.

18.1.2 Rules for capturing a global snapshot

We can use these conditions to derive some rules about how to record states. The rules will define an algorithm for capturing a consistent global snapshot.

Rule 1 *any process recording its own state will send a special marker (#) along an outgoing channel C:*

- *after recording its own state*
- *before sending anything else along channel C*

Rule 2 *The state of a channel C is recorded by process q as the sequence of messages received by q along C:*

- *after recording q's own state*
- *before receiving the marker along channel C*

In order to insure that $n \geq m$, process q shouldn't count any messages received after the marker in its own state. This leads us to the following rule:

Rule 3 *Upon receiving a marker # along on incoming channel C, a process q should do the following:*

- *case 1: q has already recorded its own state. Then q records the state of C as the sequence of messages received after it recorded its state and before it received #.*
- *case 2: q has not already recorded its own state. Then q records its own state, and records the state of C as empty.*

Rules 1-3 define an algorithm. This algorithm can start spontaneously in one or more places. A process starts the algorithm by recording its own state, then sending markers along all outgoing links. The connectivity of the network implies termination, since the markers eventually reach everywhere, and incoming markers cause a snapshot to be recorded if it hasn't already been done. The algorithm records a consistent snapshot. The snapshot is also "recent" in the following sense: it is reachable from the global beginning state of some execution sequence and the global ending state is reachable from it.

To illustrate this notion of "recent", consider the execution sequence in Figure 18.2. The execution sequence shown in the figure is as follows:

At the Beginning State, both p and q have one dollar.

Between the Beginning State and State A1:

1. p records its own state (as \$1)
2. p sends the marker to q
3. p sends a dollar to q

Between State A1 and State A2:

1. q sends a dollar to p
2. q receives the marker from p along C
3. q records its own state (as empty)
4. q records the state of C (as empty)
5. q sends the marker to p

Between State A2 and the Ending State:

1. p receives a dollar from q along D
2. p receives the marker from q along D
3. p records the state of D (as \$1)

The global snapshot is consistent in that it is a feasible state of the system, even though it *never actually arose* during the computation. The total amount of money in system (\$2) is consistent with the execution sequence. The global snapshot is also recent, as described above. We could find a total ordering of the events in the execution such that the snapshot would be ordered between the beginning and the ending state, and such that this total ordering would be consistent with the local partial order of events.

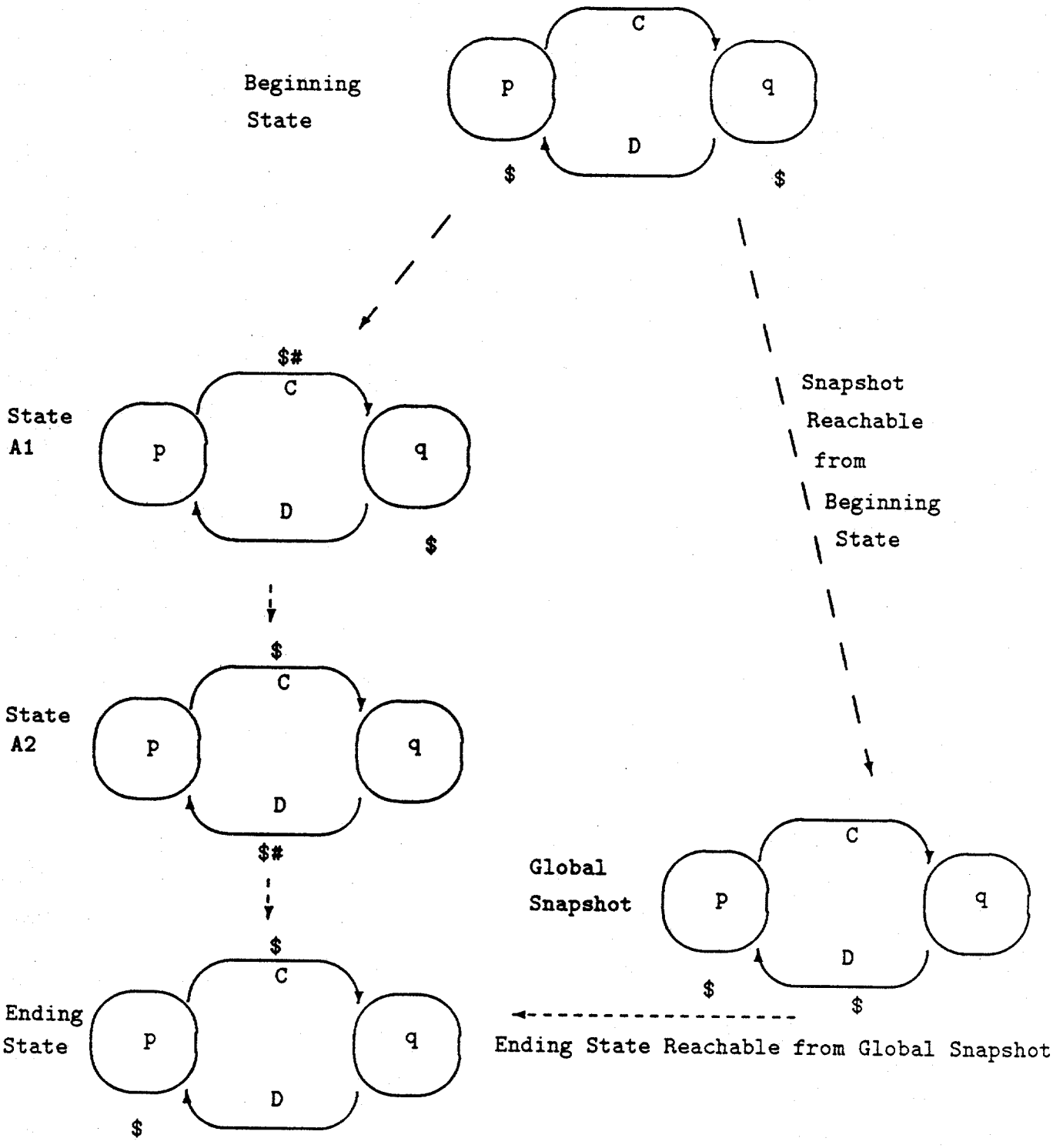


Figure 18.2: Execution sequence illustrating a recent global snapshot

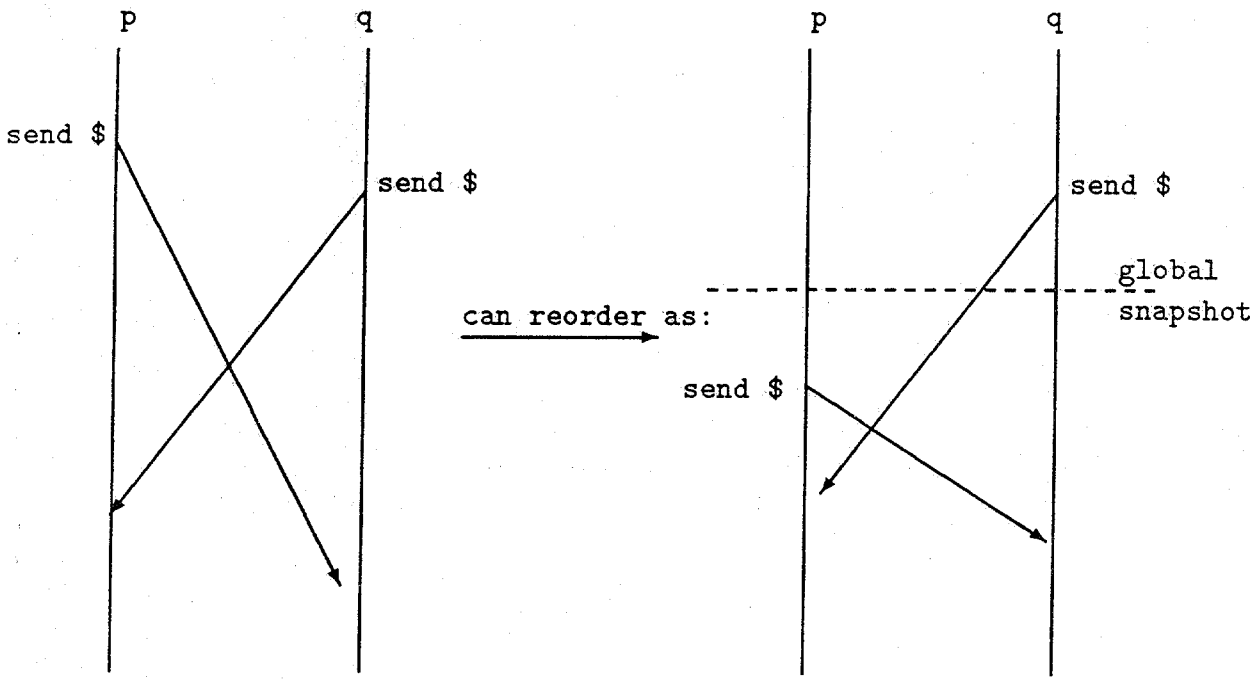


Figure 18.3: Global Snapshot as a warping of local time

A good way to think of global snapshots is as a warping of local time at each process. Although the order of global events recorded by the snapshot may not correspond to any actual state, the local view at every process is consistent with the execution sequence. This is illustrated in Figure 18.3.

18.1.3 Using global snapshots for stable property detection

An important application of global snapshots is stable property detection. Intuitively, we would like a *stable* property to be something which persists, so that a global snapshot of a system which provides information about that property at a given time can be useful at any later time.

Definition A property P is *stable* if, for all executions e, if P holds in some state R of e, then P holds in every state reachable from R.

Examples of stable properties are deadlock and termination. If we assume that processes that are waiting for resources held by other processes never give up and never release their own resources, deadlock is stable. Once a computation is deadlocked, it stays deadlocked. A deadlock detection algorithm might collect a global snapshot, then check for cycles in a

dependency graph tracking which processes are waiting for which. Similarly, a termination checking algorithm could collect a global snapshot, and check if "all processes are idle and all channels are empty".

Lemma 18.1 *If a stable property holds in a global snapshot, then it will hold at the end of a global execution sequence.*

Proof: If a stable property holds in the global snapshot, it will hold in every state reachable from that snapshot by the definition of stable. Since a global snapshot must be recent, the global ending state is reachable from it by definition. ■

Lemma 18.2 *If a stable property doesn't hold in a global snapshot, it doesn't hold at the beginning of a global execution sequence.*

Proof: The reasoning is similar to above: If the property was true at the global beginning state, it would be true in the global snapshot, since the snapshot is reachable from the beginning state. ■

18.2 Consensus Problems in the presence of faults

The problem of consensus in the presence of faults has two roots: distributed database (commit) protocols, and real-time processing. In commit protocols, processes must reach agreement about whether to make the result of a computation permanent or to abort. In real-time processing, there may be a set of sensors, some faulty, and we wish to agree on a majority value. The Byzantine agreement problem is both important and famous, and there are numerous variants, such as clock synchronization in the presence of faults, approximate agreement, etc. The first important consensus problem posed in the literature was in [Gray78], with distributed database systems as the application.

We are concerned with correct operation even when there are failures in the system. Byzantine agreement uses the most general model of node failures; anything can go wrong with a faulty node. These may be communications failures, such as the loss of messages, site failures such as the loss of a process, or malicious failures in which multiple processes may collude in the worst possible way (often called Byzantine failures). Continued operation in the face of these failures is important for many real systems.

Agreement is very easy if all processes and the communications network are reliable, but the problem becomes hard when failures are allowed. This is true even when the only failures are lost messages.

18.2.1 An impossibility result for agreement with lost messages

Consider, for example, two armies trying to surround an enemy army. If both attack simultaneously, they win, but if they attack separately, they lose. The only communication available to the generals is by runners carrying messages, and these runners may be captured behind enemy lines. Then there is no strategy allowing the generals to synchronize their attack.

An informal proof that no such strategy exists is as follows: Let α be the shortest possible protocol which solves the armies' problem. Then the last message must be necessary, or α could be shortened by deleting it. However, if this last message is lost, then a different decision must be made, so that the protocol doesn't work in the presence of lost messages.

We can show this more formally as follows:

Assume two processes, p and q, operating with a comm timebase. Without loss of generality, we will allow them perfectly synchronized local ks, and a common starting time 0. Each general (process) starts with an initial indepe "opinion". Any algorithm must meet a *validity condition*: if both generals have the sam ion NO, and if no messages get through, neither will attack. (This is a very weak conditio Any correct algorithm must also meet an *agreement condition*: if either process decides YES, the other must decide YES as well. Finally, to preclude a trivial solution in which all processes simply decide not to march, we require that there be some situation in which the decision agreed upon is to march.

Our model of a protocol is that of two synchronous automata p and q with send and receive actions. There are two possible initial states $(1, 0)$ for each automaton, corresponding to "march" and "don't march". At some point in the protocol, either process can output the decision to "march" $(= 1)$. Our correctness conditions can be more formally stated as:

- *agreement*: If either p or q outputs 1, then so does the other.
- *validity*: If both p and q have initial state 0 and no messages are delivered, then neither automaton outputs 1. There is some execution of the protocol in which both p and q output 1.

Theorem 18.3 *There is no correct agreement protocol in which the automata can output 1 in the presence of arbitrary loss of messages.*

Proof: Assume that such a protocol exists. Then consider some execution α in which the automata output 1. Let R be the last receive action by the first automaton which outputs 1 (p), as shown in Figure 18.4.

We can construct another execution α' in which no messages are received after R This is shown in the second execution of Figure 18.4. Since this execution looks identical to p, it will output 1. By the agreement condition, q must output 1 as well.

We can construct yet another execution α'' in which the receive action R is missing. Since this execution looks unchanged to q, it will output 1. Then p must output 1 as well.

By continuing this process of throwing away receive actions, we eventually reach the point at which no messages are communicated. Both automata output 1 in this execution as well, since at every step they are bound by consistency with the previous step and the agreement condition.

This execution α_k could still be valid, since we do not know what the initial condition of the processes is. However, we could construct another execution $\alpha_{k'}$ in which the initial state of q is 0. Since this execution looks identical to p, it will still output 1. Then q must output 1 as well.

In turn we can construct a final execution $\alpha_{k''}$ in which the initial states of both p and q are 0. In this execution, both processes will still output 1, since the execution looks the same as $\alpha_{k'}$ to q.

However, this last execution violates the validity condition, since neither automata receives anything, and they both output 1 even if they began with the initial condition 0. This is a contradiction. ■

18.2.2 The Byzantine agreement problem

Another failure model often used is one in which communication is reliable (perhaps through the use of timeouts, retransmissions, etc.), but some processes are maliciously faulty. We will consider this model, called the *Byzantine Agreement Problem*, over the next several lectures; [DolevS83,LamportSP82] give relevant results.

Assume a fully connected synchronous network of processes $p_1, p_2, ..., p_n$. Each process p_i has some initial value v_i, and all processes want to agree on some value. We will examine the case where communication is reliable, but processes may be faulty.

There are three conditions necessary for the protocol to work correctly:

- *agreement*: If any process decides on V, then no process decides any value but V.
- *validity*: If all processes start with initial value V, then V is the only allowable decision.
- *termination*: All processes decide.

With no failures, this is a simple problem; we can use a majority function. However, we want to tolerate *Byzantine failures*, in which faulty processes could do anything, including sending contradictory information, not sending messages, etc. Without a bound on the number of failures, it is impossible to prove anything. If we assume that we have $\leq t$ *traitors* with malicious behavior, then the problem is still much more difficult than the no-failure case. We must first refine the correctness conditions to provide for faulty processes.

- *agreement*: If any nonfaulty process decides on V, then no nonfaulty process decides any value but V.

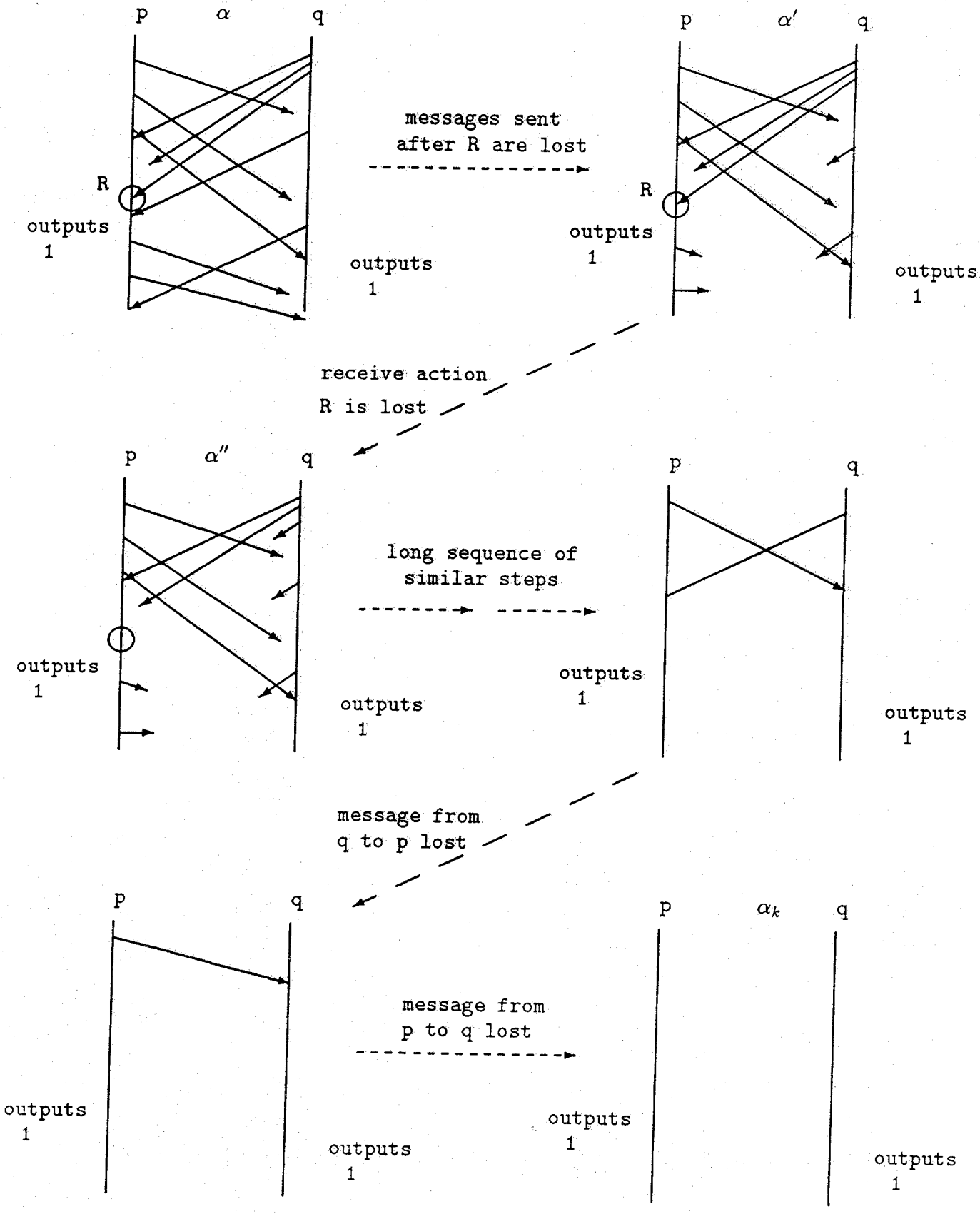


Figure 18.4: Constructing a contradictory execution by throwing away messages

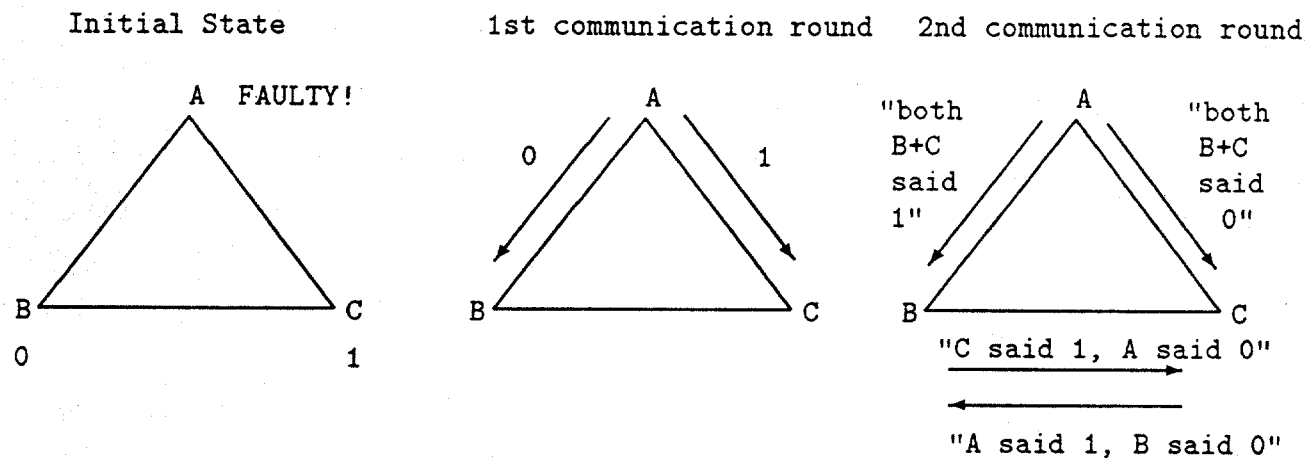


Figure 18.5: trio of processes with one fault

- *validity*: If all processes start with initial value V, then V is the only allowable decision by a nonfaulty process.
- *termination*: All nonfaulty processes decide.

As an example of why this problem is difficult, consider an example with three processes, as shown in Figure 18.5. Assume that we have one fault ($t = 1$), and the protocol goes through two rounds of communications. More than two rounds of communication buys us nothing, since we can only communicate our value, or the value that some other process sent us.

We wish to carry out some decision rule, but there can be no correct decision rule. If process A is faulty, it could send different values to the other processes in the first round, while processes B and C send their correct values. In the second round, A might in turn lie about what the processes said to it.

From process B's point of view, a configuration in which A is nonfaulty with value 0 but C is maliciously faulty is indistinguishable from a configuration in which C is nonfaulty with value 1 but A is faulty. C must decide the same way in either case, but in either case, C's decision might violate the validity condition.

A process with Byzantine faults is like a worst-case adversary. Although randomized algorithms can be used to solve Byzantine agreement with high probability, no deterministic algorithm can solve Byzantine Agreement when more than a third of the processes are faulty. To show this, we first need to prove that three processes cannot tolerate one fault, as suggested in the example above; a formal proof of this is given in [PeaseSL80]; this proof follows that in [FischerLM86].

Lemma 18.4 *Three processes cannot solve Byzantine Agreement in the presence of one fault.*

Proof: Assume they can. Then there exist three processes, A, B, and C which, when arranged in a system, run with arbitrary inputs and satisfy the Byzantine Agreement conditions even if one process malfunctions.

We could take two copies of each process, and arrange them as shown in Figure 18.6. When configured in this way, the system appears to every process as if it is configured in the original three-process system. When started up with the given inputs, they will have some behavior; we want to show that this behavior violates the correct-BA assumption.

Consider the processes A-B-C-A′. The two different processes A and A′ could send different messages to B and C. To B and C, it appears as if they are running in a three process system A-B-C, in which A is faulty. This is an allowable behavior for Byzantine Agreement, on three processes, so B and C would eventually agree on 0 in the three-process system. Since the six-process system S appears identical to them, B and C will eventually agree on 0 in S as well.

Next consider the processes A′-B′-C′-A. By similar reasoning, B' and C' will eventually agree on 1 in S.

Finally consider the processes B-C-A′-B′. To C and A′, it appears as if they are in a three-process system with B faulty. By our initial hypothesis, C and A′ must eventually agree. Although there is no requirement on which value they agree upon, either value they agree upon causes a contradiction with the value that one of the processes must agree upon with other processes. Thus there is a contradiction, and there can be no solution to the Byzantine Agreement problem for three processes when one is faulty. ■

We can use this theorem to show a more general result for Byzantine Agreement [LamportPS82]. We will do this by constructing a three-process solution from an n-process solution. If there were such a solution, we could solve the 3-process problem, which we know is impossible.

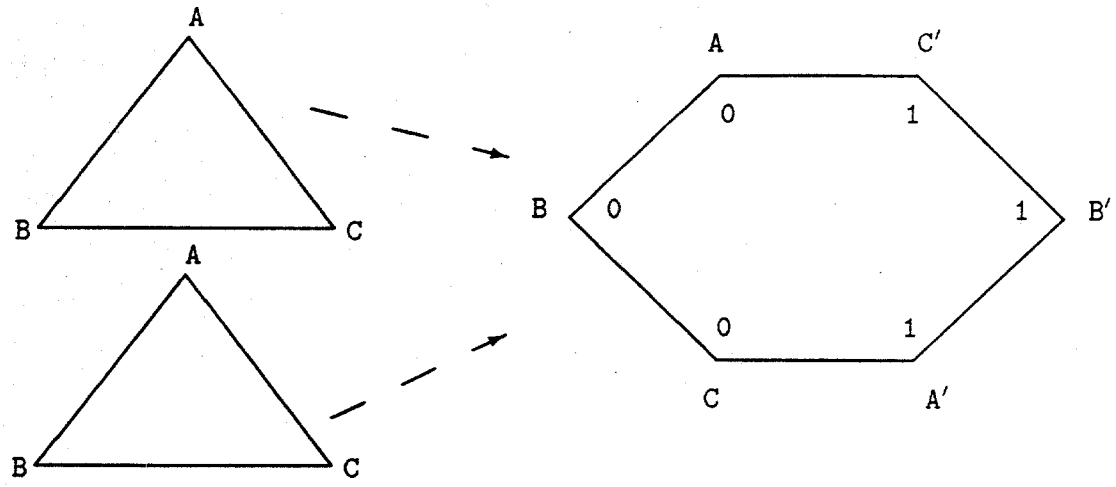


Figure 18.6: Impossibility result for Byzantine agreement on three processes with one fault

Theorem 18.5 *There is no solution to the Byzantine Agreement problem on n processes in the presence of t arbitrary node failures, when* $1 < n \leq 3t$.

Proof: Assume there is a solution for Byzantine Agreement with $3 < n \leq 3t$. (For n=2, there can be no agreement since each process has no defense against the possibility that the other could be lying).

Construct a three-process system with each new process simulating approximately one-third of the original processes. This can be done by partitioning the original processes into three subsets, P_1, P_2, and P_3, each of size s, where $1 \leq s \leq t$. Each simulating process p_i keeps copies of the states of all of the original processes in subset P_i, assigns its own initial value to every member of the set, and simulates the steps of all the processes in P_i and the messages between the processes in the subset. Messages from processes in P_i to processes in another subset are sent from P_i to the process simulating that subset.

This simulation is a protocol between three processes, p_1, p_2, and p_3. Only one of these processes is allowed to be faulty, and each simulates between 1 and t original processes, so no simulation contains more than t simulated faults. The two non-faulty processes, running a simulation correctly, can then gain agreement using the solution for $1 < n \leq 3t$. But they could use this as a solution to the 3-process Byzantine Agreement problem; this is a contradiction. ■

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 19: November 17

Lecturer: Nancy Lynch *Scribe: Jeff Palmucci*

Last lecture, we presented a lower bound of $3t + 1$ on the number of processes needed in a Byzantine Generals problem with t faulty processes. Although is not immediately obvious that Byzantine agreement can be achieved at all, an algorithm presented in [LamportSP82], can be shown to implement Byzantine Generals with $3t + 1$ processes, confirming that the bound is tight.

19.1 The Broadcast Problem

Lamport, Pease, and Shostak consider a variant of the Byzantine Generals problem, called the *broadcast problem.* In the broadcast problem we have a set of processes, $p_1, ..., p_n$, in which there may be t faulty processors. We designate a particular process, p_s, as the sender, and assign it a start value, v. The broadcast problem has conditions of agreement, validity, and termination that closely resemble those for the Byzantine Generals problem:

- **agreement:** If one non-faulty process halts with a value, v', then no non-faulty processes may halt with a value other than v'.
- **validity:** Given a non-faulty sender, no non-faulty receiver will halt with a value other then the sender's start value v.
- **termination:** All non-faulty processes eventually decide on a value.

It is possible to use an arbitrary solution to this broadcast problem to construct a solution to the Byzantine Generals problem. The idea is that each process, in parallel, will use an algorithm that solves the broadcast problem to broadcast its own value. At the completion of these algorithms, each non-faulty process will have a value, say v_i, that it received from each p_i. The solution to the Byzantine Generals problem is then defined by a selection function on these values, say $s(v_1, ..., v_n)$. By the agreement condition of the broadcast problem, each v_i is guaranteed to be the same for each non-faulty processor. Therefore, the result of the selection function will be the same for each non-faulty process, satisfying the agreement condition for the Byzantine Generals problems. We assume that the selection function will be able to satisfy the validity condition of the Byzantine Generals problem because at least $2t + 1$ values are generated from non-faulty processes.

19.1.1 A Basic Solution to the Broadcast Problem

A recursive solution to the broadcast problem is presented in [LamportSP82]. In discussing the solution, we will refer to a family of algorithms, $B(n,t)$, as follows. If n is the number of processes involved in the algorithm, and t is the number of faulty processes, we require $B(n,t)$ to have the following properties:

1. if $n \geq 3t+1$ then the properties of the broadcast problem are satisfied.
2. For any k, if $n \geq 2k+t+1$, if we have at most k failures, and if the sender is non-faulty then all non-faulty processes eventually halt with the sender's value.

The first condition simply states that $B(n,t)$ solves the broadcast problem for $n \geq 3t+1$. The second condition is an extra constraint that will be used as a stepping stone in the proof of correctness. Consider the following algorithm for $B(n,t)$:

Code for $B(n,0)$

1. The sender uses its own value.
2. It sends this value to all other processes, and they use it too.

Code for $B(n,t), t \geq 1$

1. The sender, p_s, uses its own value.
2. It sends this value to all other processes.
3. Each other process, p_i runs $B(n-1,t-1)$ to get all processes except p_s to agree on a value for p_s. (Process p_i uses the value received from p_s as its initial value, or a default if no value was received.)
4. Each p_j then considers the set of values which it decided on in the $n-1$ subcalls, using defaults for missing values. If a majority exists for some value p_j chooses it; otherwise p_j chooses the default.

We now show that Condition 2 (above) holds for this family of algorithms.

Lemma 19.1 *If the sender is non-faulty, there are at most k faults, and $n \geq 2k+t+1$, then $B(n,t)$ will terminate with each non-faulty process choosing the sender's value.*

Proof: We will prove this theorem by induction on t. The base case $B(n,0)$ is easily seen to satisfy the condition. Since the sender is non-faulty, it will send its start value, v, to each receiver, and halt with v. Each non-faulty process will correctly receive and halt with that value.

Now, assuming the lemma holds for $t-1$, we show that it holds for t. Since the sender is non-faulty, we know that it uses its own value, v, and sends v to all other processes in Step 2. Since $n \geq 2k+t+1$, we can subtract 1 from both sides to get $(n-1) \geq 2k+(t-1)+1$. Therefore, by the induction hypothesis, for every non-faulty process, p_j that is a sender in a $B(n-1,t-1)$ protocol in Step 3, we know that all the other non-faulty processes participating in that protocol must decide on the value sent by p_j. Since p_j is non-faulty, it must have sent the value v it received in Step 2.

So, it remains to be shown that enough of the processes that are senders in Step 3 are non-faulty, so that a majority function in Step 4 yields v. We derived above that $n-1 \geq 2k+1$. Therefore, since there are at most k faults, the non-faulty processes form a majority. ∎

Using this lemma, we can now go on to show that $B(n,t)$ solves the broadcast problem.

Theorem 19.2 *$B(n,t)$ solves the broadcast problem with n processors and t faulty processors, if $n \geq 3t+1$.*

Proof: The base case, $t=0$, is easy: Since there are no faulty processes, the sender will send its value to each of the receivers. The receivers, being non-faulty, will halt with this value.

For $t>0$, we assume the theorem holds for $B(n-1,t-1)$. There are two cases, either the sender is faulty, or the sender is not faulty. If the sender is non-faulty, we have a special case of Lemma 19.1. Let $k=t$. Since $n \geq 3t+1$, we have $n \geq 2k+t+1$, satisfying the preconditions of the lemma. Applying the lemma, we get that each non-faulty process terminates with the sender's value.

In the case of a faulty sender, we need only argue agreement and termination. Since the sender is faulty, we know that at most $t-1$ receiving processes are faulty. Therefore we have $n-1 \geq 3(t-1)+1$ and by the induction hypothesis, the calls to $B(n-1,t-1)$ in Step 3 will result in agreement by all non-faulty processes. This means that every non-faulty process will have the same vector of values. The majority function will result in agreement by all non-faulty processes. ∎

Since we have shown that $B(n,t)$ solves the broadcast problem, we can construct a Byzantine Agreement algorithm by the method outlined at the beginning of Section 19.1.

19.1.2 Authenticated Algorithms

One of the factors that makes the broadcast problem difficult is the ability of a faulty process to incorrectly relay information from a non-faulty process. If we can prevent this, then the solutions to the problem can be drastically simplified.

In [LamportSP82] and [PeaseSL80], a model is considered in which each process has a unique unforgeable signature, which can be validated by anyone. As values are passed around in the system, each process that sends the value applies its signature function to the message. Authenticated messages are represented by $v : s : i_1 : i_2 : ... : i_k$, where v is the value of the message, s is the signature of the original sender, and $i_j, 1 \leq j \leq k$ are the signatures of the processors that have retransmitted v.

It is easy to see that any process receiving such a message can reliably determine its authenticity. All it must do is validate the signature s. Since only one process has the ability to put s in the message, if s is valid then the message originally came from the process with signature s. As a result of this property, we refer to algorithms of this sort as *authenticated algorithms.*

A Simple Authenticated Solution

[LamportSP82] and [PeaseSL80] proposed a fairly simple broadcast algorithm with the authentication assumption. This algorithm can satisfy agreement, validity, and termination with n processors and at most t faulty processes, where $n \geq t + 1$. This improves on the $n \geq 3t + 1$ lower bound of the previous algorithm.

In the following algorithm, we denote the original sender as S, its signature as s, and its value as v_s. Each receiving process, p_i, has two local variables: V_i holds a subset of the possible values of v_s, and $accepted_i$ holds a set of authenticated messages. Initially, both V_i and $accepted_i$ are empty.

We assume that a predetermined default value is known by all non-faulty processes.

Definition A message is *valid* for p_i at round k if it is of the form $v : s : i_1 : ... : i_{k-1}$, where s and each i are valid, distinct signatures, and p_i's signature is not among them.

Code for Authenticated Byzantine Agreement

Before the start of the algorithm, S decides on its value v_s.

- Code for round 1:
  1. S transmits $v_s : s$ to all other processes.
  2. For every i, $accepted_i \leftarrow$ set of messages received by p_i that are valid for p_i at round 1.
  3. For every i, for each message m in $accepted_i$, the value of m is added to V_i.
- Code for rounds 2 through t+1:
  1. For every i, for each message m in $accepted_i$, p_i signs m and retransmits it to all other processes.

2. For every i, $accepted_i \leftarrow$ set of all messages received by p_i that are valid for p_i at this round.
3. For every i, for each message m in $accepted_i$, the value of m is added to V_i.

After the last round, each p_i examines the values of V_i. If V_i is a singleton set, p_i decides on the value in this set. Otherwise, p_i decides on the predefined default.

Theorem 19.3 *The Lamport-Pease-Shostak authenticated algorithm solves Byzantine agreement.*

Proof: Termination is obvious, since the algorithm halts immediately after round $t+1$.

Consider the case where S is non-faulty. Every non-faulty process, p_i, accepts $v_s : s$ at round 1, and puts v_s into V_i. Since no other process can produce s, no valid messages will contain a value other than v_s. Therefore, upon termination, each $V_i = \{v_s\}$, and each non-faulty process decides v_s. Therefore, agreement and validity hold.

In order to show agreement in the case where S is faulty, we will argue that upon termination, $V_i = V_j$ for all non-faulty i, j. If $v \in V_i$, then let m be the first message accepted by p_i that contains v. If m arrives before or at round t, then p_i will sign and relay the message to p_j. In the next round, p_j will accept the message and add it to V_j. If m arrives in round $t+1$, we know that t receivers have previously accepted it. Since the sender is faulty, there are at most $t-1$ faulty receiving processes. Therefore, at least one of the signatures in m is for a non-faulty process, which must have also sent m to p_j. Since $V_i = V_j$ for all non-faulty i, j, each decides on the same value. ■

Achieving Polynomial Communication

The message complexity of the above algorithm is exponential in t. We shall now examine an algorithm that achieves polynomial message complexity, [DolevS82].

The Dolev-Strong algorithm is a simple modification of the authenticated algorithm proposed above. A process, p_i, upon accepting a message $v : s : i_1 : \ldots : i_k$, will only rebroadcast that message if adding v to V_i will result in $|V_i| \leq 2$. In other words, each non-faulty process will only sign and relay the first 2 accepted messages that have distinct values.

Theorem 19.4 *The Dolev-Strong authenticated algorithm solves the broadcast problem using $O(n^2)$ messages.*

Proof: For the message complexity, we note that any message sent by p_i after the first round results from an addition to the set V_i. Since $|V_i| \leq 2$ for all i, each p_i will send at most 2 messages to each p_j. This gives a message complexity of $O(n^2)$.

Now, on to the the proof of correctness. Termination is obvious, since the algorithm will halt after $t+1$ rounds. When the sender S is non-faulty, the arguments for agreement and

validity are the same as for the [LamportSP82] algorithm. Therefore, we can move directly on to the case of a faulty sender.

In proving agreement when S is faulty, we first show that after round $t+1$, $V_i = \{v\} \Rightarrow V_j = \{v\}$. First, assume $V_i = \{v\}$ and fix some j. We first show that $v \in V_j$. Let m be the first message that arrives at p_i with value v. If p_j has signed m, then $v \in V_j$. If m has not been signed by p_j, it is necessary to consider two cases. If m arrives at p_i before round $t+1$, then p_i will sign and relay m to p_j. Then p_j will accept v, resulting in $v \in V_j$. If m arrives at round $t+1$, we know that t receivers have previously accepted it. Since the sender is faulty, at most $t-1$ of these receivers could be faulty. Therefore, by the pigeonhole principle, at least one of these is non-faulty, and must have also sent m to p_j. Therefore p_j must accept v by round $t+1$ and $v \in V_j$.

Now we must show that no value other than v can be in any V_j. Assume, for contradiction, that V_j contains some value other than v. Let m be the first message with value $w \neq v$ accepted by any non-faulty process p_j. Arguing as before, either (1) p_i's signature is in m, so $w \in V_i$, or (2) p_i's signature is not in m and m arrives at p_j before or at rount t, so p_j relays m to p_i (since w is at most its second value), resulting in $w \in V_i$, or (3) p_i's signature is not in m and m arrives at p_j at rount $t+1$, so some non-faulty process signed and sent m to p_i, resulting in $w \in V_i$. In all three cases $w \in V_i$, a contradiction.

After round $t+1$, each p_i will terminate with the default if $|V_i| > 1$. If $|V_i| \leq 1$ we are guaranteed that each V is the same. In both cases, we have agreement. ■

At IBM, variants of this algorithm have been implemented in which process identifiers are used as signatures, since the faulty processors are not malicious. With a large enough id domain, it is reasonable to assume that an accidental forgery is unlikely.

19.2 Limiting Communication Cost

We have seen that the communication cost for authenticated agreement algorithms is small, but that the communication cost in the Byzantine fault model without authentication is very large (exponential in the number of faults). One way to cut down on communication cost is to reduce the lengths of the messages by cutting down the domain of possible values.

This approach was taken in [TurpinC84]. The idea is to solve Byzantine agreement for a multivalued domain by running a Byzantine agreement algorithm for a single bit as a subroutine. If the multivalued domain is large, the savings can be substantial.

The Turpin-Coan algorithm requires that $n \geq 3t+1$. A default value is known initially by all non-faulty processes. For each process, a local variable x is initialized to the input value for that process. (Notice that we are returning to the consensus version of the problem, as opposed to the broadcast version.)

- Code for round 1:
  1. Broadcast x to all other processes.
  2. In the set of messages received, if there are $\geq n - t$ for a particular value, v_{max}, then $x \leftarrow v_{max}$, otherwise $x \leftarrow nil$.
- Code for round 2:
  1. Broadcast x to all other processes.
  2. Let v_{max} be the value, other than nil, that occurs most often among those values received, with ties broken in some consistent way. Let *num* be the number of occurrences of v_{max}.
  3. if $num \geq n - t$ then $vote = 1$ else $vote = 0$.

After round 2, run the binary Byzantine agreement subroutine using *vote* as the input value. If the bit decided upon is 1, then decide v_{max}, otherwise decide the default value.

Claim 19.5 *At most one value v is sent in round 2 messages by correct processes.*

Proof: Any process p sending a value v in round 2 must have received at least $n - t$ messages containing v in round one. Since there are at most t faulty processes, this means that all other processes received at least $n - 2t$ copies of v. Since the number of messages received is n, no process could have received $n - t$ messages containing a value $v' \neq v$ in round 1. Therefore, the claim holds. ∎

Theorem 19.6 *The Turpin-Coan algorithm solves multivalued Byzantine agreement when given a boolean Byzantine agreement algorithm as a subroutine.*

Proof: It is easy to see that this algorithm terminates.

To show validity, we must prove that if all non-faulty processes start with a value, w, then all non-faulty processes must decide w. After the first round, all non-faulty processes will have set $x \leftarrow w$ because at least $n - t$ processes broadcast it reliably. Therefore, in the second round, each non-faulty process will have $v_{max} = w$, $num \geq n - t$, and $vote = 1$. The binary agreement subroutine is therefore required to choose 1, and each non-faulty process will choose w.

In showing agreement, there are two cases. If the subroutine decides on $vote = 0$, then the default value is chosen by all non-faulty processes, so agreement holds. If the subroutine decides on $vote = 1$, then we must argue that the local variable x is the same for each non-faulty process. Note that for the subroutine to agree on $vote = 1$, then some non-faulty process p must have started with $vote = 1$. Therefore, process p must have received at least $n - t$ round 2 messages containing some value v. Since there are at most t faulty

processes, each other process must have received at least $n - 2t$ round two messages with non-nil values from non-faulty processes. By Claim 19.5, *all* of the non-nil messages from non-faulty processes must have the same value, namely v. Therefore, v must be the value occurring most often, since there are at most t faulty processes and $n - 2t > t$. ∎

19.3 Subsequent Results

There has been a subsequent series of papers giving polynomial communication algorithms for Byzantine Agreement. We will not give the details here, but only summarize the results.

Polynomial communication and $2k+t$ rounds (for constant k), assuming $3t+1$ processes, was achieved without cryptographic assumptions in a brute force algorithm by Dolev, Fischer, Fowler, Lynch, and Strong [DolevFFLS82].

These ideas were rethought by Srikanth and Toueg [SrikanthT87], who essentially substituted an authentication protocol for the assumed signature scheme in the Dolev-Strong algorithm. Using this authentication protocol requires twice the number of rounds as does the Dolev-Strong algorithm, and also needs $3t+1$ processes. The basic idea is that whenever a message was sent in the Dolev-Strong algorithm, Srikanth and Toueg run a protocol to send and accept the message.

An improvement on the $2t$ rounds required by the above algorithms was achieved by Coan [Coan86], who presented a family of algorithms requiring $t+\epsilon t$ rounds, where $0 < \epsilon \leq 1$. The message complexity is polynomial, but as ϵ approaches zero, the degree of the polynomial increases. This result implies that no lower bound bigger than $t+1$ rounds can be proved for polynomial algorithms, although no fixed degree polynomial algorithm is actually given for $t+1$ rounds. A paper by Bar Noy, Dolev, Dwork, and Strong [BarNoyDDS87] presents these ideas in a different way.

Finally, a new paper by Moses and Waarts claims to achieve $t+1$ rounds with polynomial communication [MosesW88].

19.4 More on the Numbers of Processes

We have shown that Byzantine agreement can be solved with n processes and t faults, where $n \geq 3t+1$. In proving this result, we assumed that any process could send a message directly to any other process. We now consider the problem of Byzantine agreement in general communication graphs [Dolev82].

Consider a communication graph, G, where the nodes represent processes and an edge exists between two processes if they can communicate. It is easy to see that if G is a tree, we cannot accomplish Byzantine agreement with even one faulty process. Any faulty process that is not a leaf would essentially cut off one section of G from another. The non-faulty processors in different components would not be able to reliably communicate, much less

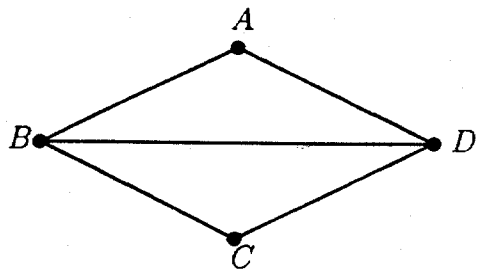


Figure 19.1: A graph G with $conn(G) = 2$.

reach agreement. Similarly, if removing t nodes can disconnect the graph, it should also be impossible to reach agreement with t faulty processes.

Definition The *connectivity* of a graph G, $conn(G)$, is the minimum number of nodes whose removal results in a disconnected graph. We say that a single node graph has a connectivity of 1. Furthermore, we say a graph G is *k-connected* if $conn(G) \geq k$.

Figure 19.1 shows a graph with a connectivity of two. If B and D are removed, then we are left with two disconnected pieces, A and C.

Our proof for the lower bound on connectivity for Byzantine agreement uses methods similar to those used in our upper bound proof for the number of faulty processes. Recall the technique of joining up arbitrary processes to appropriately-named neighbors, such that the resulting configuration must do *something*. Also, recall the following two axioms.

- **Locality Axiom:** A process's actions depend only on messages from its input channels and its initial value.
- **Fault Axiom:** A faulty process is allowed to exhibit any combination of behaviors on its outgoing channels, provided that the behavior of each channel can arise in some system in which the process is acting correctly.

The locality axiom basically states that communication only takes place over the edges of the graph, and thus it is only these inputs and a process's initial value that can affect its behavior. The fault axiom expresses a masquerading capability of failed processes. We cannot determine if a particular edge leads to a correct process, or to a faulty process simulating the behavior of a correct process over the edge. The fault axiom gives faulty processes the ability to simulate the behaviors of different correct processes over different edges.

With these basic concepts, we can now prove a lower bound on connectivity for solving Byzantine agreement.

Theorem 19.7 *It is possible to solve Byzantine agreement on a graph G, with n nodes and t faults if and only if $n \geq 3t+1$ and $conn(G) \geq 2t+1$.*

Proof: We already know that $n \geq 3t+1$ processes are required. For a fully connected graph. It is easy to see that this situation will not improve for an arbitrary communication graph.

We start by showing that Byzantine agreement is possible if $conn(G) \geq 2t+1$. (The *if* direction.) *Menger's Theorem* states that a graph is k-connected if and only if every pair of points is joined by at least k node-disjoint points. Since we are assuming G is $2t+1$-connected, there are at least $2t+1$ node disjoint paths between any two nodes. We can simulate a direct connection between these nodes by sending the value along each of the $2t+1$ paths. Since only t processes are faulty, we are guaranteed that the value received in the majority of these messages is correct. Therefore, simulation of a fully connected graph can be accomplished.

We now prove the *only if* direction of the connectivity argument. The argument that Byzantine agreement is not possible if $conn(G) < 2t+1$ is a bit more intricate. We will first take $t = 1$, for simplicity.

Assume there exists a graph, G, with $conn(G) \leq 2$ which can solve Byzantine agreement with one fault. Two points in G can disconnect the graph. The graph in Figure 19.1 can be generalized to any graph with a connectivity of 2 by replacing B and D with arbitrary graphs. To keep our argument simple, however, we will consider B and D to be single nodes. We can construct a graph C by "rewiring" two copies of graph G, as shown in Figure 19.2. Each process in C behaves as if it was the same-named process in Figure 19.1 with the input denoted by the subscript.

Consider the behavior of the processes outlined in Figure 19.3, and the corresponding behavior in 19.4, where F is a faulty process. Since F is allowed to simulate any graph, the outlined processes cannot tell the difference between Figure 19.3 and Figure 19.4. Therefore, by the validity property, these processes must all decide 0. Now consider Figure 19.5 and the analogous situation in Figure 19.6. By the same argument, all the outlined processes are required to decide 1. Finally, consider Figure 19.7, which is analogous to the situation in Figure 19.8. Since only F is faulty, the agreement condition requires that the outlined processes decide on the same value. However, we have already shown that process A_1 must decide 1 and process C_0 must decide 0. Thus, we have reached a contradiction. It follows that we cannot solve Byzantine agreement for $conn(G) \leq 2$ and $t = 1$.

To generalize the result to $t > 1$, we use the same diagrams, with B and D replaced by graphs of at most t nodes each. Again, removing B and D disconnects A and C. The edges of Figure 19.1 now represent all possible edges between A, B, C, and D. ∎

The bounds of $n \geq 3t+1$ and $conn(G) \geq 2t+1$ carry over to a wide variety of consensus problems. [FischerLM85] show that these bounds also hold for problems such as weak Byzantine agreement, Byzantine firing squad, approximate agreement, and clock synchronization.

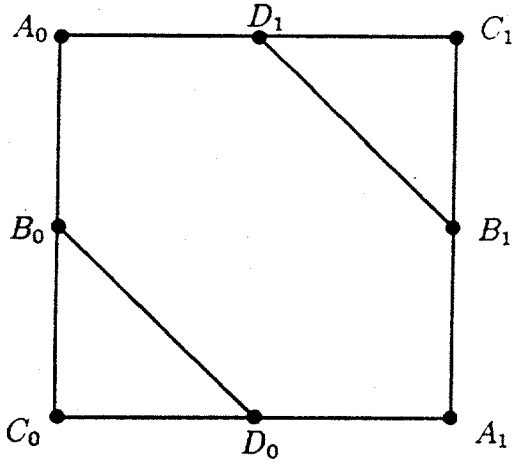


Figure 19.2: Graph C, made by "rewiring" two copies of G.

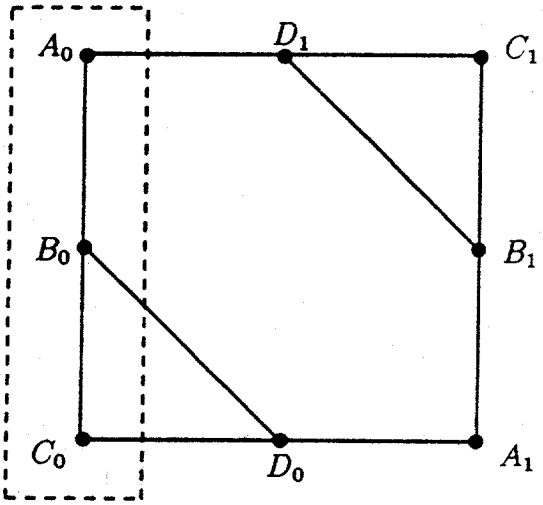


Figure 19.3: A set of processes in C.

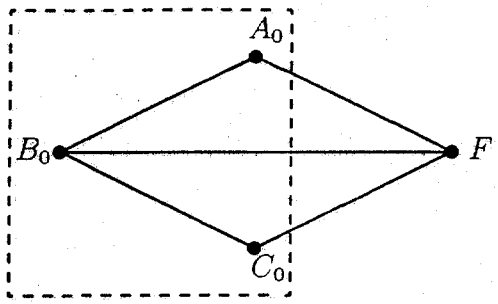


Figure 19.4: A configuration (with F faulty) that the outlined processes cannot distinguish from Figure 19.3.

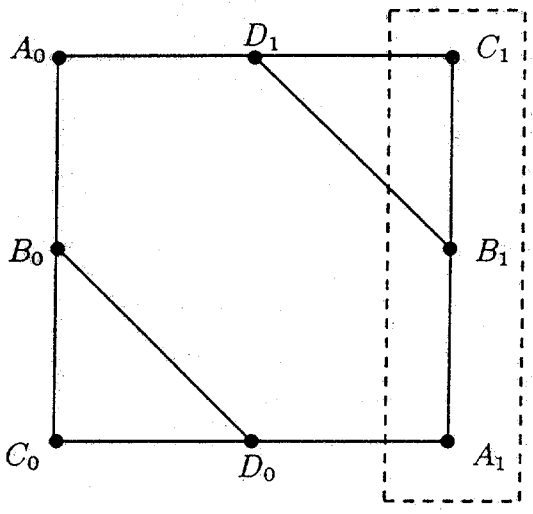


Figure 19.5: A set of processes in C.

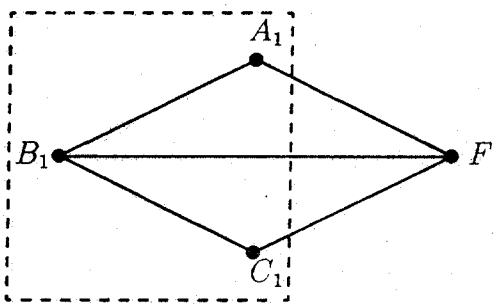


Figure 19.6: A configuration (with F faulty) that the outlined processes cannot distinguish from Figure 19.5.

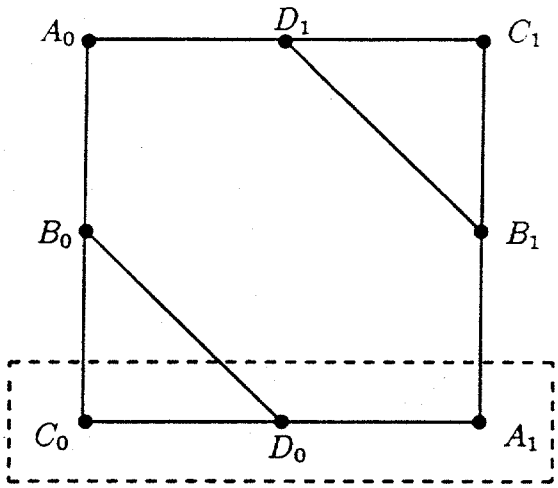


Figure 19.7: A set of processes in C.

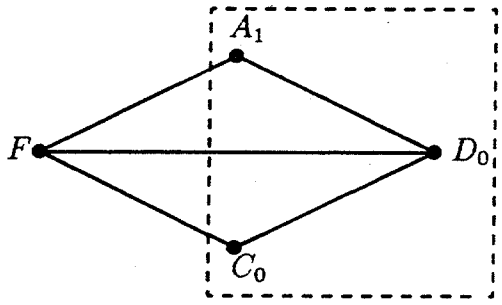


Figure 19.8: The non-faulty processes must agree, giving us a contradiction.

6.852 Distributed Algorithms **Fall Semester, 1988**

Lecture 20: November 22

Lecturer: Nancy Lynch *Scribe: George Varghese*

In the previous lecture we looked at synchronous algorithms to solve the Byzantine Agreement (BA) problem. It so happened that all the algorithms we saw used $t + 1$ rounds of communication to reach consensus, with at most t faulty processes. This suggests the question: can we find a faster synchronous algorithm that solves BA in t or fewer rounds?

The answer to this question is basically no, and was established in a series of papers. The fundamental result was proved by Fischer and Lynch [FischerL85] for the symmetric version of BA assuming that processes could not use authentication. Later papers showed that the lower bound holds even if processes are allowed to use authentication [DolevS82], and even if failures are restricted to weaker stopping failures [MosesT88].

In this lecture we will examine only the basic Fischer-Lynch result and the extension by Merritt [MosesT88] to the case of stopping failures. An important theme of this lecture will be the use of chain arguments to establish impossibility results.

20.1 Using chain arguments for Impossibility Results

In a sense lower bounds and impossibility results are labor saving devices: they prevent fruitless effort. But they also capture the essence of why a problem is hard. Any effort to do better must change some assumption under which the lower bound was proved – for instance by introducing randomization.

All the impossibility and lower bound results we have seen arise from a property of distributed algorithms: the action taken by a process in the algorithm is determined by what the process knows locally. Given this, an impossibility proof for a distributed algorithm often runs as follows.

First we assume that the distributed algorithm can be solved subject to the specified cost constraints. The distributed algorithm is unspecified, except for the result it must produce. Next we choose a set of executions of the distributed algorithm. For each execution, we know what a local process must compute based on what the process sees locally and what the result of the distributed algorithm must be. Finally, we show that, when taken together, the values computed in each instance lead to a contradiction.

In this lecture the structure of the impossibility proofs will be in the more specific form of a "chain argument". Suppose we model the behaviour of a process in the distributed algorithm by a function F. A chain is a chain of equalities on the value of F on different input i.e.

$F(I_1) = F(I_2) = F(I_3) = ... = F(I_n)$

Each equality in the chain (i.e. $F(I_j) = F(I_{j+1})$ is established by considering an execution (or executions) of the distributed algorithm with process inputs I_j and I_{j+1} that must (by the problem definition) produce the same result. Finally we obtain a contradiction by showing separately that $F(I_1)$ cannot equal $F(I_n)$. Note that we typically use *different* executions to establish each link in the chain, but the function stays the same.

In this lecture we will see three chain arguments – two in the Fischer-Lynch result, and one in the Merritt result.

20.2 The Fischer-Lynch Lower Bound

Theorem 20.1 Any *synchronous algorithm that solves the symmetric version of BA without using authentication must take at least $t+1$ rounds, if there are t faulty processes.*

The proof consists of three parts: a model of any synchronous algorithm that solves BA, an intermediate Lemma that shows there is no loss in generality by assuming that all non-faulty processes follow the same algorithm; and finally the main proof.

20.2.1 Model used in Fischer-Lynch Lower Bound

We first describe a way to model *any* synchronous algorithm that solves the BA problem under the given assumptions. Each non-faulty synchronous process is modelled by an automaton that has:

- A Message Generation Function to decide what messages to send to other processes based on its state.
- A State Transition Function that determines a new state based on previous state and incoming messages.

Faulty processes do not follow these functions and can exhibit arbitrary behaviour.

The execution is synchronized. On each round each process sends messages to other processes. Each process then computes its new state based on all received messages before the next round starts. (Note: We cannot model these process automata as I/O Automata since I/O automata essentially operate asynchronously; we could, however, model the whole collection of processes as one big I/O automaton, but we do not do so in what follows.)

Because the algorithm does not use authentication, we argue that we can cast *any* synchronous algorithm that solves BA in a "normal form". This simplifies the notation and what we have to reason about. Informally, in normal form each process always broadcasts "everything it knows" to every other process.

More precisely, the behavior of each process p_i is:

- First round: Broadcast initial value.
- Second round: Broadcast what everyone told p_i their initial values were.
- Third round: Broadcast what everyone told p_i that everyone told them their initial values were.

. . . and so on

Given normal form, we can use a simple notation for the information sent in messages on each round:

- First round: X (i.e. my value is X)
- Second round: Xp (i.e. p told me X)
- Third round: Xpq (i.e. q told me p told him X)

. . . and so on

Assume also that processes send messages to themselves just as they do to other processes – e.g. process q sends Xp to itself, then Xpq to itself etc. This simplifies things nicely because the information available to process q after round i is captured by the set of messages that process q receives in round i. Thus process q's actions in round i only depend on the messages it receives in round i.

This produces an exponential number of messages (N^i messages for N processes and i rounds). However, we are interested only in time bounds. What we are saying is that even given as much message complexity as it can possibly use (as expressed by normal form), no algorithm can beat the lower bound on time of $t + 1$ rounds.

Why is it the case that we can transform any algorithm that solves BA (under the given assumptions) to normal form? After all, a particular algorithm may choose not to send certain pieces of "what it knows" or even send "only some function of what it knows". However, we can simulate all such algorithms by appropriate changes to the process state transition functions while preserving normal form. We see this informally by considering the following cases:

- Algorithms in which processes send some other function of the message information in each round: Instead of computing the function at the sender, send all possible information in normal form and compute the function at the receiver.
- Algorithms in which processes don't send some information in some rounds: instead, send all possible information in normal form, and ignore appropriate pieces of information at the receiver.

- Algorithms in which non-faulty processes determine that some processes are faulty, and choose not to send information to faulty processes: In this case, modify the faulty process functions (which are arbitrary) to guess anything that a non-faulty process does not send it in a round. Because we are doing a worst case analysis, we can confine ourselves to executions in which these guesses are always correct!

A careful proof of this model's generality can be found in [FischerL85].

Given this model, we now have a simple representation of a process's state. After k rounds, a process's state is simply a k-dimensional array A (each dimension of length n) of initial values. If your state is represented by the array A, then $A(i_1, i_2, ..i_k)$ is the value that process i_k told you process i_{k-1} told him . . . process i_2 told him was process i_1 's initial value. Another recursive way of looking at this is that A consists of n subarrays of dimension $k-1$, each of which are the states of every other process after the $k-1$st round. The second view explains how the state of a process is updated after each round: each non-faulty process broadcasts its state after $k-1$ rounds to all other processes in round k. In what follows we will sometimes refer to an instance of the array A as a "view".

Finally, if the protocol terminates after r rounds, then there must also be a function F_p for each process p, that maps every view A to a value V. The value V is the value decided by process p at the completion of the Byzantine Agreement protocol. Note that F_p can be derived from the message sending and state transition functions of the automaton.

20.2.2 Intermediate Lemma for Fischer-Lynch lower bound

Now it is possible that each process p has a different F_p. The intermediate lemma below simplifies the reasoning needed for the final proof by showing that wlog all functions F_p can be considered the same.

Lemma 20.2 *Let $n \geq 2t+1$. In a synchronous BA algorithm with t faults, we have $F_p(A) = F_q(A)$ for all processes p and q and all views A such that:*

1. *A is the view of p in some execution in which p is non-faulty and there are no more than t faults.*
2. *A also is the view of q in some execution in which q is non-faulty and there are no more than t faults.*

Proof: First there is no loss in generality in assuming $n \geq 2t+1$ because in the previous lecture we proved that *any* solution of BA (regardless of time constraints) must have $n \geq 3t$.

Next, the Lemma is trivially true if the two executions referred to in the Lemma are the same. (In that case p and q are non-faulty in the same execution and must decide the same value by the Agreement property.)

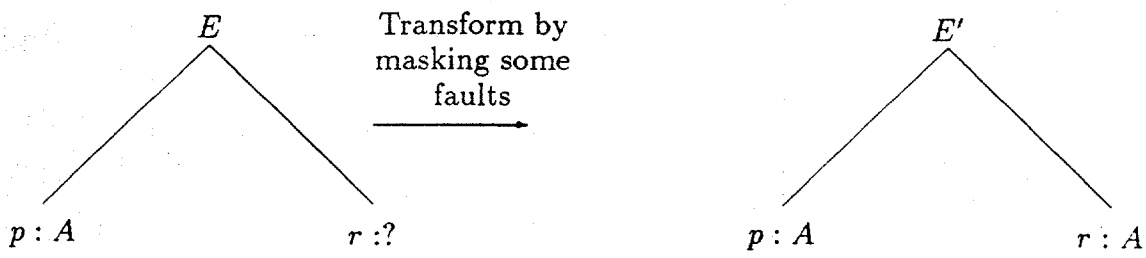


Figure 20.1: Transforming one execution into another by masking faults

The difficult part is to show the Lemma to be true if A occurs in some execution E in which p is non-faulty and also in another execution G in which q is non-faulty. The proof is by the chain argument we referred to at the start.

First, we show that there must be at least one process r that is non-faulty in both E and G.

The number of faulty processes in E is no more than t. The number of faulty processes in G is no more than t. Thus the number of processes that are faulty in either E or G is no more than $2t$. Since n is at least one greater than $2t$, there must be at least one process r that is not faulty in either E or F. This is the only place where we need $n \geq 2t + 1$.

We now use process r as a "bridge" between executions E and G. To do so we transform E into another execution E' which is identical to E except that in E' process p and process r get exactly the same messages in the last round. This is illustrated in Figure 20.1.

We claim that E' is a valid execution of the algorithm (with less than t faults and process p and r non-faulty) just as E is. Why is this true? Suppose in E, processes p and r receive different information from process f in the last round. Then since p and r are non-faulty, f must be faulty. But since f's behaviour is arbitrary we can always consider another valid execution (i.e. E') in which f sends the same information to r as it does to p. Proceeding in this way, we can transform E to E' by "masking" all faults in the last round of E.

Thus in E' both process p and r see the same view A. Thus by Byzantine agreement we must have $F_p(A) = F_r(A)$.

We do the same thing to execution G i.e. transform it into another valid execution G' in which process q and r both see the same view A. By Byzantine agreement this gives us $F_r(A) = F_q(A)$

Together the two equalities give us our first example of a chain, a chain of size two. (The chains get bigger as the later proofs become more intricate.):

$$F_p(A) = F_r(A) = F_q(A)$$

■

This intermediate lemma allows to drop the subscript and assume there is a single function F used by all non-faulty processes to decide a value based on their views. This simplifies our reasoning and notation in the main proof.

20.2.3 Proof of Fischer-Lynch lower bound

Proof: Assume for simplicity that the value V to be agreed upon by the processes is either 0 or 1. Assume we have an algorithm that reaches consensus in t rounds, with t faulty processes.

Again our strategy is to find a chain of equalities on the function F that leads to a contradiction. We start by picking the end points of the chain.

If a process p has a view A consisting of all 0's then p's decision must be 0. This is because process p could obtain this view if all processes were non-faulty and started with a 0. Thus by the validity condition, process p must decide on 0. Denote the view consisting of all 0's by A_0. Then $F(A_0) = 0$.

By similar arguments, if we denote the view consisting of all 1's as A_n then $F(A_n) = 1$.

Then we can get the required contradiction if we can produce a chain of equalities leading to $F(A_0) = F(A_1)$ i.e. $F(A_1) = F(A_2) = ...F(A_k) = F(A_{k+1}) = ...F(A_n)$.

To establish each equality in the chain, we have to show that any two consecutive views A_i and A_{i+1} appear as the views of two different non-faulty processes in the same t round execution with no more than t faulty processes. Then by agreement we must have $F(A_k) = F(A_{k+1})$. As usual we will establish each equality by a different execution.

To understand how this is done, we first examine the chain for the special case of $t = 2$; the extension to general t is considered later.

For $t = 2$, after the second round, each view must be a two dimensional $n \times n$ matrix, where $A[i, j]$ is the value that j told you that i said his initial value was.

Consider a general link in the chain that establishes that $F(A_k) = F(A_{k+1}), 0 \leq k < n$. A_k and A_{k+1} are identical except for position i, j . Our goal is to construct a single execution in which one non-faulty process will "see" view A_k and a second non-faulty process will "see" view A_{k+1}. This is shown in Figure 20.2.

To do so, consider an execution E(k) with the following properties:

1. All processes except i and j are non-faulty.

2. All processes less than i have initial value 0.

3. All processes greater than i have initial value

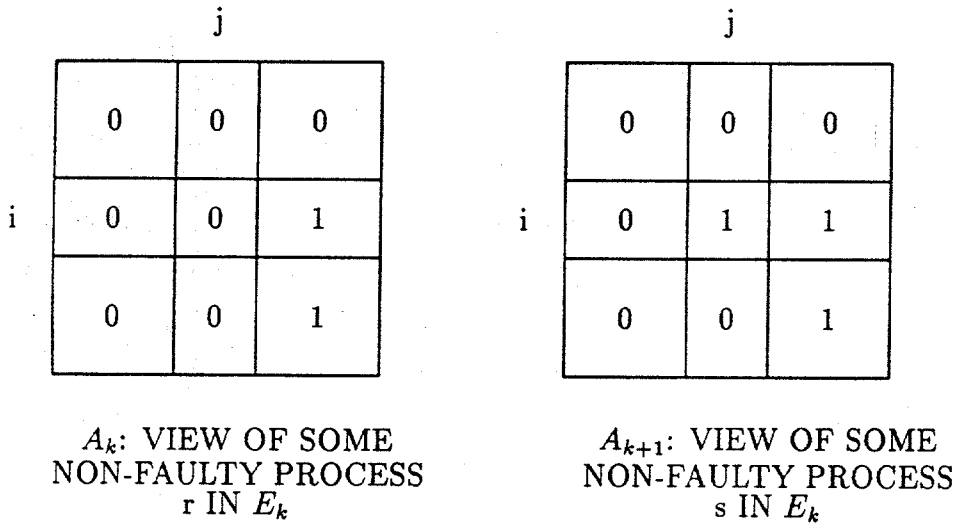


Figure 20.2: Two consecutive views used to establish a link in the chain

Since all non-faulty processes send and relay correctly, all non-faulty processes will receive views identical to A_k and A_{k+1} except for the ith row and jth column.

To fill in most of the jth column assume that process j (perversely) relays every value it receives correctly except for the value received from process i. Thus all non-faulty processes have the same jth column in their views except for element (i, j).

To fill in most of the ith row, assume that i sends 0 to all processes $< j$ and 1 to processes $> j$. The value i sends to j doesn't matter because j is faulty. Thus all non-faulty processes will have the same ith row in their views except for element (i, j).

Taken together, the constructions in the last three paragraphs imply that all non-faulty processes see exactly the same views except for element (i, j).

We now apply the *coup de grace*. Process j tells one non-faulty process (say r) that i told him 0, and then process j tells a second non-faulty process (say s) that i told him 1. This fills in Element (i, j) differently for process r and process s. Process r sees view A_k while process s sees view A_{k+1}. But process r and process s are non-faulty in the same execution. Hence by agreement: $F(A_k) = F(A_{k+1})$.

We have now established a generic link in the chain. To get the entire chain (that links a view with all 0's to one with all 1's) we use $n^2 - 1$ intermediate views. Each intermediate view has an extra 1 over the previous views. This can be done systematically by working in row order ("creeping across rows") changing one bit at a time.

The generalization to more than two dimensions (i.e $t > 2$) is similar: once again we work in lexicographic order converting 1's to 0's. Now the chain will be longer (there are now n^t 1's we need to change to 0's) and we will need t faults to establish a generic link in the chain. ■

20.3 Impossibility Result for Stopping Faults

We have proved that any synchronous algorithm that achieves distributed consensus in the presence of up to t processes with Byzantine faults must take at least $t + 1$ rounds. Surprisingly, the result is still true if process failures are restricted to simpler stopping failures – i.e. processes can only fail by stopping completely. The theorem was first described in [MosesT88] based on unpublished work by Merritt (based in turn on preliminary results by Hadzilacos and Fischer-Lamport).

Theorem 20.3 *In the crash model (i.e. processes can only fail by stopping), consensus requires $t + 1$ rounds in the worst case, with t faulty processes.*

To establish an impossibility result, a theoretician is almost in the position of an adversary who draws upon possible failure modes to "attack" the result and get a contradiction. While simpler failure modes make life easier for protocol designers, they make it harder to establish impossibility results.

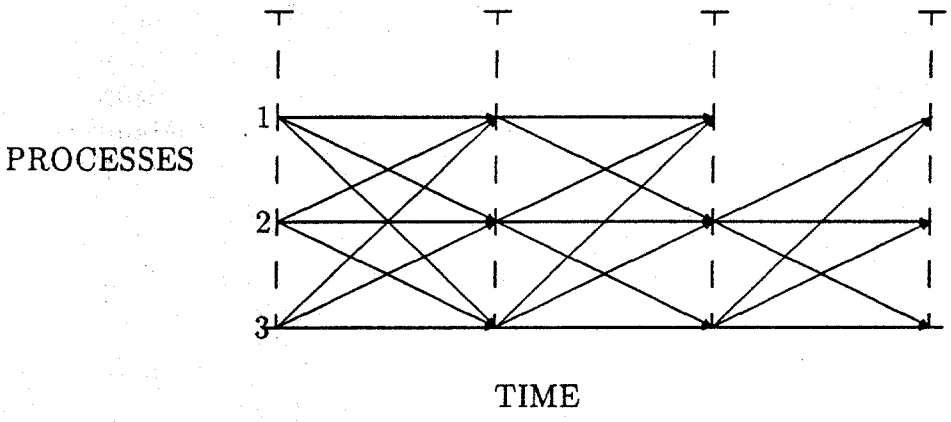


Figure 20.3: Example of a communication pattern

For instance, in the previous proof we used a simplifying reduction to normal form. Part of the justification for this was that faulty processes could guess any values they were not sent. Clearly we cannot use normal form if processes are restricted to stopping failures. Similarly, the main proof happily used faulty processes that sent conflicting information. Because this "power" is no longer available to us, the proof is more intricate and the chains are longer.

Once again the proof has three parts: a model, a proof based on an intermediate lemma, and finally the proof of the lemma.

20.3.1 Model and Notation

We assume a protocol in which each process sends information to every other process at every round. All processes start with initial values; eventually a process may write "decide V" in its state. But for simplicity, we assume a process continues executing forever even after it decides. As usual each process that is alive has a deterministic state transition function.

Define an *execution* as an infinite sequence of tuples. The ith tuple in the sequence contains all the process states at time i and all the messages sent in the ith round.

Define a *communication pattern* of an execution as some representation of which processes send to which other processes in each round. A communication pattern does not tell us the actual information sent but only "who sent to whom" in a round. A communication pattern can be depicted graphically as shown in Figure 20.3.

In the figure, p_1 does not send to p_3 in round 2. Thus p_1 must have stopped and will send nothing further in round 3 and future rounds. Essentially a communication pattern depicts how processes fail in a run.

Given the initial state tuple and the communication pattern, we can determine an execution uniquely from the (deterministic) process state transition functions.

Define a *run* as an initial state tuple plus the communication pattern. Denote by $exec(\rho)$ the execution generated by run ρ.

A process is faulty in a run or an execution exactly if it stops sending somewhere in the communication pattern. There are never more than t faulty processes in a run.

We write $(\rho, l) \overset{p}{\sim} (\rho', l)$, (where ρ and ρ' are runs, l is a time, and p is a non-faulty process in both ρ and ρ') to mean that $exec(\rho)$ and $exec(\rho')$ are indistinguishable to process p through time l. That is, process p sends and receives the same messages and has the same state tuple sequence in executions ρ and ρ' up through time l.

We write $(\rho, l) \sim (\rho', l)$ if there is some non-faulty process p in both ρ and ρ' such that $(\rho, l) \overset{p}{\sim} (\rho', l)$ – that is the two executions look the same to some non-faulty process up through time l.

We write $(\rho, l) \approx (\rho', l)$ for the transitive closure of $\sim$. That is there exists some sequence of runs such that $(\rho, l) \sim (\rho_1, l) \sim (\rho_2, l) \sim ... \sim (\rho', l)$.

Finally, analogous to the function F we used in the previous proof, we define a decision function $dec(\rho, t)$ that when applied to a run produces the value decided by a process at time t.

20.3.2 Proof using an Intermediate Lemma

Proof: Suppose we had the following lemma to work with.

Lemma 20.4 *Let ρ and ρ' be two runs, each with $\leq f$ faulty processes, where $f \leq t$. Let p be some process and assume that ρ and ρ' only differ in p's failure behavior after time k. If $l - k \leq t + 1 - f$ then $(\rho, l) \approx (\rho', l)$.*

This lemma can be used to show it is impossible to reach consensus in t rounds. Assume we had such a protocol and that 1 and 0 are the only possible initial values. Then we arrive at a contradiction as follows.

The strategy is to set up a chain starting from a run that starts with all 0's and has no failures (and hence must decide 0) to a run that starts with all 1's and has no failures (and hence must decide 1). We first see how the first link in the chain is set up.

We know that every (ρ, t) has a corresponding decision value dec (ρ, t). If $(\rho, t) \sim (\rho', t)$ then $dec(\rho, t) = dec(\rho', t)$ (since some non-faulty process has the same view and uses the same decision function in both executions). Thus if $(\rho, t) \approx (\rho', t)$ then $dec(\rho, t) = dec(\rho', t)$

Let ρ_0 and ρ' be two runs that both start with 0's. In ρ_0, no one fails. In ρ_0', process p_1 fails at time 0, and sends no messages. Now apply the lemma to these two runs with $f = 1$

(only 1 failure), $p = p_1$ (only process whose behaviour is different in the two runs), $k = 0$ (time after which the runs look different), and $l = t$. With these values $l - k \leq t + 1 - f$ because (by plugging in values) $t - 0 \leq t + 1 - 2$. Thus the lemma tells us that $(\rho_0, t) \approx (\rho_0', t)$

Let ρ_0'' have 1000...0 as input (i.e. the initial value of process 1 is a 1, while that of all other processes is a 0) and where process 1 fails at time 0. Clearly $(\rho_0', t) \approx (\rho_0'', t)$ since the only change is not seen.

Let ρ_1 have 1000...0 as input but no one fails. By applying the Lemma again (and using the same values of f, p, k, l etc.) to the two runs ρ_1 and ρ_0'' we get $(\rho_0'', t) \approx (\rho_1, t)$. The last three equalities taken together imply that $(\rho_0, t) \approx (\rho_1, t)$. Thus we have set up the first link in the chain which shows that the value of the decision function on failure-free runs starting with 000..0 and 100..0 must be same.

In general, let (ρ_i, t) for $2 \leq i \leq n$ be a failure-free run in which the first i processes get 1's and the remainder get 0's, By using exactly the same technique we used to establish $(\rho_0, t) \approx (\rho_1, t)$, we can show in general that $(\rho_{i-1}, t) \approx (\rho_i, t)$ This, of course, sets up a chain which leads to $(\rho_0, t) \approx (\rho_n, t)$, which in turn implies that $dec(\rho_0, t) = dec(\rho_n, t)$. But this contradicts validity: the left hand side must be 0, as it is the value decided when all processes get initial value 0 and no process fails; the right hand side must be 1 as it is the value decided when all processes get initial value 1 and no process fails. ∎

20.3.3 Proof of Intermediate Lemma

We now prove the Intermediate Lemma. Refer back to the statement of Lemma 2 in the previous section.

Proof: If $k \geq l$, then the two runs ρ and ρ' only differ in their behaviour after time l, and must be indistinguishable up to time l. In this case the Lemma is trivially satisfied.

So assume $k > l$. We will use backwards induction on $j = l - k$.

The lemma is also trivially satisfied if process p is non-faulty in both ρ and ρ'. So assume that process p is faulty in one of the two runs. Each of ρ and ρ' has $\leq f$ faults. Also, if p is non-faulty in one run, then it must have $\leq f - 1$ faults in that run. (If it had f faults, the other run would have $f + 1$ faults because p must be an additional faulty process in the other run.)

Induction Basis: $l - k = 1$.

ρ and ρ' agree up to time k. Consider round $k + 1$. Thus up to the lth round, the two runs only differ in whom p sends to on round $k + 1 (= l)$.

Since we know that $n \geq 2t + 1$ there must at least $t + 1$ non-faulty processes in ρ. Of these, we pick two non-faulty processes in ρ, say q and r.

First construct a new run ρ_r that is identical to ρ except that p sends to r exactly if p sends to r in ρ'. Next construct a second new run ρ_r' that is identical to ρ except that p sends to exactly the same processes in round $k + 1$ that p sends to in ρ'. An example of how

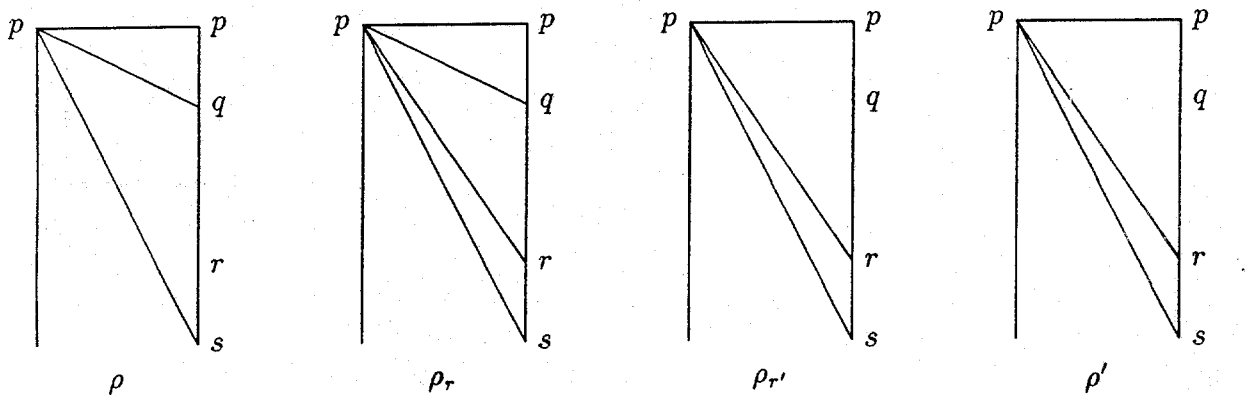


Figure 20.4: Example of the first set of constructions used to prove Lemma 20.4

ρ_r and ρ'_r are constructed from ρ and ρ' is sketched in Figure 20.4. Each of the four figures in Figure 20.4 represent a communication pattern in round $k+1$.

Then ρ_r has $\leq f$ failures and (by construction) ρ and ρ_r are indistinguishable to process q up time l. Thus $(\rho, l) \sim (\rho_r, l)$. Also, ρ'_r has $\leq f$ failures and (by construction) ρ'_r and ρ_r are indistinguishable to process r up time l. Thus $(\rho_r, l) \sim (\rho'_r, l)$. Also, ρ'_r is clearly identical (by construction) to ρ' up to time l. Thus $(\rho'_r, l) \sim (\rho', l)$.

Thus the last three identities set up a chain that links ρ and ρ'. This leads to $(\rho, l) \approx (\rho', l)$ which proves the lemma.

Inductive Step: $l - k > 1$

Let $\rho_i, 1 \leq i \leq n$, be the same as ρ except in the messages process p sends to the first i processes in round $k+1$. The difference is that in round $k+1$, process p sends to $p_1, ...p_i$ exactly as it does in ρ'.

We will first prove that $(rho_i, 1) \approx (\rho_{i+1}, 1), 1 \leq i < n$. We observe that:

1. ρ_i and ρ_{i+1} differ at most in what process p sends to process p_{i+1} in round $k+1$.
2. At most f processes fail (if we are to apply the lemma) in both ρ_i and ρ_{i+1}.
3. We know that $1 < l - k$ (assumed for inductive step) and that (if we are to apply the lemma) $l - k \leq t + 1 - f$. These two inequalities together imply that $f < t$.

This suggests constructing two intermediate processes ρ'_i and ρ'_{i+1}. Let ρ'_i be identical to ρ_i except that process p_{i+1} is silent (i.e. dies) after time $k+1$. Similarly, let ρ'_{i+1} be identical

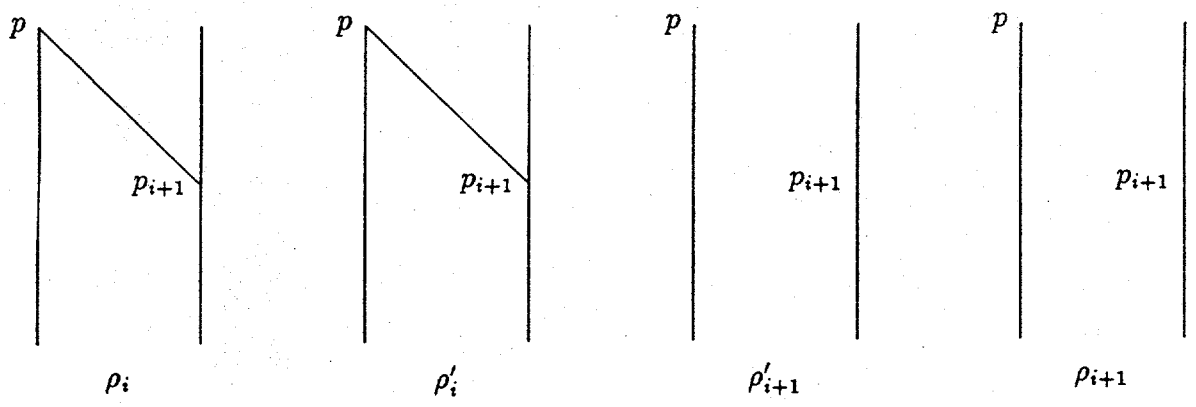


Figure 20.5: Example of the second set of constructions used to prove Lemma 20.4

to ρ_{i+1} except that process p_{i+1} is silent after time $k+1$. Note that by the third observation, adding an extra failure to ρ_i and ρ_{i+1} does not cause the total number of failures to exceed t.

An example of these constructions is sketched in Figure 20.5.

We observe that:

1. ρ_i and $\rho_{i'}$ differ only in the behaviour of process p_{i+1} after time $k+1$.
2. Less than $f+1$ processes fail in ρ_i and $\rho_{i'}$.
3. $l-k+1 \leq t+1-(f+1)$ since $l-k \leq t+1-f$.

Thus we can apply the inductive hypothesis to conclude that $(\rho_i, l) \approx (\rho'_i, l)$. A corresponding argument shows that $(\rho_{i+1}, l) \approx (\rho'_{i+1}, l)$. But clearly $(\rho_{i+1}, l) \approx (\rho'_{i+1}, l)$ since the two runs look identical to any non-faulty process. (The only way they can differ is in terms of messages sent to p_{i+1} in round $k+1$; but p_{i+1} is silent after round $k+1$.) Together these three equations yield a generic link in the chain: $(\rho_i, l) \approx (\rho_{i+1}, l), 1 \leq i < n$.

The transitive closure of this relation gives us $(\rho, l) = (\rho_0, l) \approx (\rho_n, l)$ But $(\rho_n, l) \approx (\rho', l)$. This follows from the inductive hypothesis, since the two runs differ only in the failure behaviour of process p after time $k+1$; also $l-(k+1) \leq t+1-f$ if $l-k \leq t+1-f$. These last two equations give us $(\rho, l) \approx (\rho', l)$ which completes the proof of the lemma. ∎

20.4 Exercises

1. Can the Chandy-Lamport global snapshot algorithm be described as a collection of new "filter" automata, one per node of the original system?

   Each filter f_i should intercept all the send and receive actions to and from the corresponding automaton P_i of the original system, and should carry on some new processing before passing the send and receive actions to their intended destinations.

   Sketch how such f_i might be constructed. If modifications to P_i or extra assumptions about P_i are needed, discuss those.

2. What happens to solvability of the two-generals consensus problem, if

   (a) messages are guaranteed to eventually arrive, but there is no bound on how long they take?

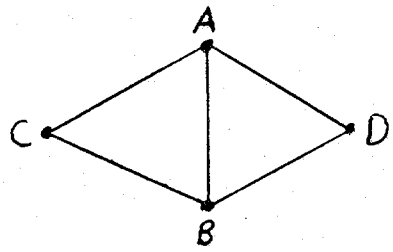
   (b) there exists a bound b such that all messages either arrive within time b or not at all?

   (c) there exists a fixed bound b such that all messages always arrive within time b (and none are lost)?

3. Analyze the communication and number of round complexities of the basic Lamport-Pease-Shostak Byzantine agreement algorithm.

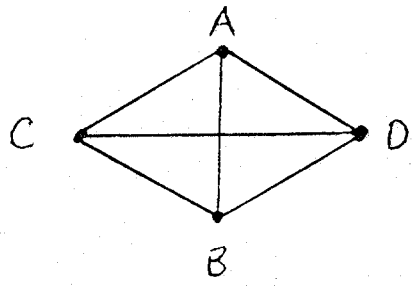
4. Suppose $n \geq 4t + 1$. Design an algorithm that uses a subroutine for binary Byzantine agreement and solves multivalued Byzantine agreement. This algorithm should improve on the Turpin-Coan algorithm by only requiring one additional round rather than two.

5. Reconsider the proof that Byzantine agreement cannot be reached in the graph:



   in the presence of one fault.

   Why doesn't the proof extend to the graph:



6. Complete the proof of the $t+1$ lower bound on rounds for Byzantine agreement. That is, for the case where $t > 2$, describe the chain of views and show any consecutive pair can be produced in a single execution with at most t faulty processes.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 21: November 29

Lecturer: Mark Tuttle | *Scribe: John Keen*

21.1 Knowledge

The intuition underlying the theory of knowledge is that a processor has some idea about its surrounding environment. We will formalize this intuitive idea of knowledge in this lecture.

The theory of knowledge will be seen to be quite useful. It will allow us to construct good (optimal, in fact) protocols for consensus, where all processors start with some input bit and they all reach agreement on some value. For such an algorithm, we want all processes to reach agreement in the same round.

21.1.1 Optimality

In the crash failure model, a run ρ is determined by three things:

1. The protocol P executed by each processor in the system.
2. The input I to each of the processors.
3. The communication graph G (which describes the pattern of process failures).

Note that in the consensus problem processors receive input only at time 0, the initial input bits, but in other problems the processors might receive input from some external source at other times as well. We now state some basic definitions.

Definition Two runs ρ and ρ' are *corresponding runs* if they have the same input and communication graphs. That is, ρ and ρ' differ only in the protocol followed being during the runs.

Definition A consensus protocol P is *optimal in all runs* if
$\forall$ P', where P' is an arbitrary consensus protocol, it is true that
$\forall \rho, \rho'$ being corresponding runs of P and P', respectively,
if the nonfaulty processes decide at time l in ρ',
then the nonfaulty processes decide no later than l in ρ.

In this definition, we measure performance (and hence optimality) only in terms of the number of rounds. This definition is a very strong condition for optimality, since it applies to *all* runs for protocol P, and not merely the worst case.

21.1.2 The Muddy Children Story

A cute story will give us a good feel for the theory of knowledge. It captures the notion of knowledge in a distributed system. The story is called *the muddy children problem.*

One rainy day, a bunch of children went outside to play. Their mother warned them to keep clean and stay away from the mud, but they foolishly ignored her and frolicked in the mud. Some of them got mud on their foreheads and were therefore dirty, but some others managed to stay clean. Eventually, their mother came out and saw that some were dirty. She told them, "Someone is dirty." She asked them, "Can you prove that you are dirty?" She continued to ask this question until all the dirty children responded "yes" in unison. That is, if there are k dirty children, all children respond "no" in the first k-1 rounds, but all dirty children answer "yes" in round k. Why does this happen?

Theorem 21.1 *If there are k dirty children, then all children will all answer "no" in rounds 1,2,...,k-1 and all dirty children will answer "yes" in round k.*

Proof: We will prove this by induction on k.

The basis is for k=1. Recall that we must have at least one dirty child, due to the statement of the problem. If there is only one dirty child, he will see that all the other children have clean foreheads. Since everyone else is clean, he must therefore be dirty and so he answers "yes".

Now consider the case for k>1. Consider one of the dirty children. We will call her Nancy, for example. Nancy can see k-1 other children that are dirty, but cannot see whether or not her forehead is dirty. She knows there can be either k-1 or k dirty children. For any round i, where i≤k-1, Nancy will respond "no", as will all the other dirty children. She must respond "no" because she cannot *prove* that she is dirty. The other children would have given the same response in the rounds preceding round i regardless of whether she was clean or dirty. But now consider round k. If Nancy were clean, there would be only k-1 dirty children, and they would have all answered "yes" in round k-1, by our inductive hypothesis. Since this did not happen in round k-1, Nancy now knows that it is not possible that she is clean. She knows she is dirty, and answers "yes" in round k, as do all the other dirty children. ∎

The moral of this entertaining story is that knowledge somehow depends on what we know is possible, given that we are in a particular state. In rounds 1 through k, Nancy does not know that she is dirty since there are two worlds, one in which she is dirty and one in which she is not, that are indistinguishable from the current state of the world. Finally, in round k, Nancy is able to eliminate the world in which she is no dirty as a possibility. Since Nancy is dirty in all worlds (in this case, the single world) that she is unable to distinguish from the actual world, she knows that she is indeed dirty. In general, if a fact is known to hold in all possible worlds that cannot be distinguished from the current state of the world, then the fact is known to be true. We will now define this formally.

21.1.3 Formal Theory of Knowledge

Consider a protocol P. Let S_P denote the set of all runs for P with all possible inputs and all possible communication graphs.

Recall that (ρ,l) denotes a *point* in a run. It represents the state of the system after the first l rounds in run ρ. We will abuse notation somewhat and write (ρ,l)$\in S_P$ iff $\rho \in S_P$.

We write

$$(\rho,l) \models \varphi \text{ iff } \varphi \text{ is true in } (\rho,l)$$

to indicate that the fact φ is considered to be true in the system if it is in state corresponding to point (ρ,l).

We define knowledge of a fact φ by a processor q when the system is at a point (ρ,l) by

$$(\rho,l) \models K_q\varphi \text{ iff } (\rho',l') \models \varphi \text{ for all } (\rho',l')\in S_P \text{ satisfying } (\rho',l')\tilde{\sim}_q(\rho,l).$$

The symbol K_q means "processor q knows". Recall that if nonfaulty processor q has the same *view* at point (ρ',l') as at point (ρ,l), then we write (ρ',l')$\tilde{\sim}_q$(ρ,l).

As a first attempt to define the notion of "everyone knows", we might say

$$E_G = \bigwedge_{q\in G} K_q\varphi$$

to mean that every processor in some set G of processors knows φ. This is a very natural definition if a processor can determine whether or not it belongs to G, but it is problematic when it cannot.

Consider any run of a consensus protocol and let φ_v denote the fact that some nonfaulty processor is deciding v. Let N represent the set of nonfaulty processors. One property it seems natural to expect runs of P to satisfy is the following:

$$\varphi_v \Rightarrow E_N\varphi_v.$$

To provide a proof for this claim, we might try to argue as follows. Suppose some nonfaulty process q is deciding v. Then all nonfaulty processors must be deciding v. Then all nonfaulty processors know that some nonfaulty processor (namely, itself) is deciding v. End of proof. But a processor doesn't know whether or not it is nonfaulty. It may know that it has not failed as of yet, but it does not know it will never fail in the future. A nonfaulty process deciding v knows only that *if* is nonfaulty, *then* there is some nonfaulty processor (itself) that is deciding v.

Because of this difficulty of a processor not knowing whether or not it is nonfaulty, we revise our first attempt given above. We state the following definition.

Definition $E_N\varphi = \bigwedge_{q\in N} K_q(q \in N \Rightarrow \varphi)$

This definition says that every nonfaulty processor knows φ iff every nonfaulty processor q knows that *if* it is nonfaulty *then* φ is true.

With this modified definition, our preceding argument proves the following:

Theorem 21.2 $\varphi_v \Rightarrow E_N\varphi_v$.

It turns out that the state of common knowledge is crucial to the construction of optimal protocols. Intuitively, a fact φ is common knowledge if φ is true, everyone knows φ, everyone knows everyone knows φ, and so on. Formally, we have the following.

Definition Common knowledge of a fact φ by the nonfaulty processors is defined as
$$C_N\varphi = \varphi \wedge E_N\varphi \wedge E_N E_N\varphi \wedge \cdots$$

We can imagine the idea of a similarity graph for the system. It serves as a useful way of thinking about common knowledge.

Definition A similarity graph can be constructed from all the points of a system as follows:
nodes: (ρ,l)
edges: (ρ,l) $\tilde{\sim}_q$ (ρ',l')
That is, the nodes are all possible points for the system, and an edge connects any two points for which the view is the same for some nonfaulty processor q.

As in the previous lecture, let us define (ρ,l)$\sim$(ρ',l') iff $\exists$ points (ρ_i,l_i) and runs q_i such that
$$(\rho,l) \tilde{\sim}_{q_1} (\rho_1,l_1) \tilde{\sim}_{q_2} \cdots \tilde{\sim}_{q_k} (\rho',l').$$
We can prove the following characterization of common knowledge.

Theorem 21.3 *$(\rho,l) \models C_N\varphi$ iff $(\rho',l) \models \varphi$ $\forall (\rho',l) \sim (\rho,l)$.*

Since (ρ,l)$\sim$(ρ',l') iff $\exists$ a path from (ρ,l) to (ρ',l') in the similarity graph, this theorem means that a fact is common knowledge at a point (ρ,l) only if it holds at all points in the connected component of (ρ,l). Some interesting properties follow from this theorem.

Property 1: If $\varphi \Rightarrow E_N\varphi$ is true, then $\varphi \Rightarrow C_N\varphi$ is true. This is proven by an induction on k that φ holds at all points of distance at most k from any point satisfying φ, and hence at all points in this point's connected component.

Property 1 allows us to prove the following claim (recall that $\varphi_v \Rightarrow E_N\varphi_v$).

Claim 21.4 $\varphi_v \Rightarrow C_N\varphi_v$.

Property 2: If $C\varphi$ and $\varphi \Rightarrow \psi$ are true, then $C\psi$ is true. The proof for this is simply that if both φ and $\varphi \Rightarrow \psi$ hold at all points in the computation, then so does ψ.

Property 2 lets us prove the following claim.

Claim 21.5 $\varphi_v \Rightarrow C_N$*(at least one input bit is v)*

As proof of this claim, recall that $\varphi_v \Rightarrow C_N\varphi_v$. The definition of consensus says $\varphi_v \Rightarrow$"$\exists$v". Thus, $\varphi_v \Rightarrow C_N\varphi_v \Rightarrow C_N(\exists \text{v})$.

We see that common knowledge is a necessary condition for making a decision for consensus.

Theorem 21.6 *Let P be an arbitrary consensus protocol. If some nonfaulty processor decides on the value $v \in \{0,1\}$ at time ℓ in run ρ of P, then $(\rho,\ell) \models C_{\mathcal{N}}(\exists v)$.*

Property 3: $C_{\mathcal{N}}\varphi \Rightarrow E_{\mathcal{N}}C_{\mathcal{N}}\varphi$

This says that as soon as a fact becomes common knowledge to the nonfaulty processors, all nonfaulty processors know the fact is common knowledge. Thus, as soon as the value of an input bit becomes common knowledge, all nonfaulty processors can decide on a value simultaneously. This suggests that we want a protocol that leads to knowledge about all the input bits becoming common knowledge as fast as possible. This leads us to define the *full information protocol* $\mathcal{F}$:

repeat every round
    send your current state to all processors
forever

We can prove the following.

Theorem 21.7 *Let P be an arbitrary protocol, and let $\mathcal{F}$ be the full information protocol. Let ρ and τ be corresponding runs of P and $\mathcal{F}$, respectively, and let φ be a fact about the input. If $(\rho,\ell) \models C_{\mathcal{N}}\varphi$ then $(\tau,\ell) \models C_{\mathcal{N}}\varphi$.*

In particular, we can take φ to be $\exists$v. We therefore claim that the following protocol $\mathcal{P}$ is the fastest protocol for consensus. That is, we claim that the protocol solves the consensus problem and is optimal in all runs.

repeat every round
    send your current state to all processors
until $C_{\mathcal{N}}(\exists 0)$ or $C_{\mathcal{N}}(\exists 1)$
if $C_{\mathcal{N}}(\exists 0)$ **then**
    decide on 0 and halt
else
    decide on 1 and halt

Here, for $v \in \{0,1\}$, we write $C_{\mathcal{N}}(\exists v)$ as shorthand for

$$C_{\mathcal{N}}(\textit{some processor's input bit is equal to } v).$$

Let us complete the proof of the claim that this protocol is the fastest protocol for consensus. Recall that A protocol P for a problem X is said to be *optimal in all runs* if the following condition holds: for every protocol P' solving X and for all pairs of corresponding runs ρ and ρ' of P and P', respectively, if the nonfaulty processors halt at time ℓ in the run ρ' of P', then the nonfaulty processors halt at time ℓ or earlier in the run ρ of P. In other words, fix the input processors receive and fix the pattern of processor failures, and run consider runs of the two protocols P and P' in this fixed setting. Processors are guaranteed to halt in the run of P at least as soon as they halt in the run of P'. Thus, not only is P optimal in the sense that it meets the worst-case lower bound in its worst-case run, it does as well in *every* run as any other protocol could possibly do.

Notice that $\mathcal{P}$ is essentially the full-information protocol $\mathcal{F}$, except that it halts as soon as one of two facts become common knowledge, rather than continuing forever. Recall also that a fact about the input is a fact that depends only on the input processors receive during the run, a fact such as "*at least four processors have input bits equal to* 1."

Now, ignoring for the moment the problem of proving that $\mathcal{P}$ actually solves the consensus problem, let us prove a result saying that $\mathcal{P}$ should be optimal in all runs.

Lemma 21.8 *Let P be an arbitrary consensus protocol. Let ρ and τ be corresponding runs of P and $\mathcal{P}$. If the nonfaulty processors decide on a value at time ℓ in ρ, then the nonfaulty processors decide on a value no later than time ℓ in τ.*

Proof: Suppose the nonfaulty processors decide on v at time ℓ in the run ρ of P. Since some nonfaulty processor is deciding on v at time ℓ in ρ, Theorem 21.6 implies that $(\rho, \ell) \models C_{\mathcal{N}}(\exists v)$. Since $(\rho, \ell) \models C_{\mathcal{N}}(\exists v)$, Theorem 21.7 implies that $(\tau, \ell) \models C_{\mathcal{N}}(\exists v)$. It follows, therefore, that all nonfaulty processors can halt at time ℓ in the run τ of $\mathcal{P}$ and decide on a value, unless, of course, they have already halted. ∎

Now let us prove that $\mathcal{P}$ is a consensus protocol optimal in all runs.

Theorem 21.9 *The protocol $\mathcal{P}$ is a consensus protocol that is optimal in all runs.*

Proof: To prove that $\mathcal{P}$ is a consensus protocol we must prove that in every run

1. all nonfaulty processors decide on a value $v \in \{0, 1\}$,
2. all nonfaulty processors decide simultaneously on the same value, and
3. if all input bits are v, then all nonfaulty processors decide on v.

To see that all nonfaulty processors do decide on a value, observe that there are consensus protocols that halt within $t + 1$ rounds in every run, and hence by Lemma 21.8 the protocol $\mathcal{P}$ also halts within $t + 1$ rounds in every run. To see that all nonfaulty processors decide simultaneously on the same value v, recall that $C_{\mathcal{N}}\varphi \Rightarrow E_{\mathcal{N}}(C_{\mathcal{N}}\varphi)$. It follows that as soon

as one nonfaulty processor decides on v, by Theorem 21.6 we have $C_{\mathcal{N}}(\exists v)$, and hence all nonfaulty processors know $C_{\mathcal{N}}(\exists v)$. It follows that all nonfaulty processors will decide on v and halt. To see that if all input bits are v then all nonfaulty processors decide on v, notice that if all input bits are v then $C_{\mathcal{N}}(\exists \bar{v})$ can never hold, and hence no nonfaulty processor will decide on $\bar{v}$. Since all nonfaulty processors must eventually decide, they must decide on v. Finally, the fact that $\mathcal{P}$ is optimal in all runs follows directly from Lemma 21.8. ■

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 22: December 1

Lecturer: Nancy Lynch *Scribe: Andrew Sutherland*

22.1 Consensus in Asynchronous Systems

Up to this point the consensus problems we have considered have all presumed some level of synchronization between the processes. We now consider the more general problem of consensus in a completely asynchronous distributed system. No assumptions are made about the relative speeds of the processes or about the length of any delays in message delivery. In particular, we assume that it is not possible to distinguish between a process that has halted and one that is merely running very slowly (or is experiencing a very long delay in receiving a message). It is assumed that all processes execute in a deterministic fashion—randomized solutions will be examined in a future lecture.

Under these conditions, we find the surprising result that consensus is not possible. Even if we restrict failures simply to stopping faults (i.e. Byzantine types of failures are disallowed) and assume a completely reliable message passing system, the possibility of the failure of even a single process precludes any solution to the consensus problem. This result has far-reaching implications, and at the end of this section a similar argument is used to show the impossibility of constructing atomic *test-and-set registers* from atomic *read-write registers*. (This result was claimed without proof in Lecture 14.)

22.1.1 The Consensus Problem

Assume that every process starts with an initial value from $\{0,1\}$. A process *decides* on a value in $\{0,1\}$ by entering an appropriate decision state. A process *fails* by halting (i.e. not taking any more steps). The requirements for a solution are as follows:

1. *Agreement*: No two non-faulty processes may decide on different values.

2. *Validity*: If all non-faulty processes have the same initial value, then no other value may be decided upon by a non-faulty process.

3. *Termination*: All non-faulty processes must eventually decide.

22.1.2 Modeling the System

We will use I/O automata to model the asynchronous system, as shown in Figure 22.1. (This presentation is somewhat simpler than the presentation in the original paper [FischerLP85].) Each process p_i, $1 \leq i \leq n$, is modelled as an I/O automaton with the following restrictions for simplicity:

- All state transitions are deterministic. That is, for any state s of p_i and action π there is at most one transition (s, π, s').
- For each initial value (in $\{0,1\}$), p_i has a unique start state.
- There is exacly one equivalence class in p_i's partition.[1]

The message system is modelled by a particular I/O automaton as follows. The state of the message system is a multiset of (m,i) pairs, where m is a message from some universal set of messages and $1 \leq i \leq n$. Each input action to the message system is of the form $bcast_i(m)$ (an output of p_i) and results in the insertion of the n pairs (m,j), for all $1 \leq j \leq n$, into the multiset. Each output action of the message sytem is of the form $receive_i(m)$ (an input to p_i), is enabled whenever (m,i) is an element of the multiset, and results in the deletion of that pair from the multiset. Each $receive_i(m)$ action is in its own class of the partition. In this way, a fair execution of the message system must have every message sent eventually being delivered.

Note that there are no internal actions of the message system. Thus, every step of the message system involves exactly one process p_i. Furthermore, the structure of the system ensures that any step is a step of only one process since all interactions between processes must occur via the message system.

Definition A *1-fair execution* is an execution in which the message system and all but possibly one process continue to take locally controlled steps. (This corresponds to the possible stop-fault of a single process.)

Definition A *0-resilient consensus protocol* (0-RCP) is a protocol that solves the consensus problem in the absence of faults—it must solve consensus for all fair executions. Similarly, a *1-resilient consensus protocol* (1-RCP) is a protocol that solves the consensus problem in the presence of at most one stop-fault—it must solve consensus for all 1-fair executions. Note

[1] If an algorithm makes use of countably many equivalence classes, then it may be simulated with a single class by dovetailing.

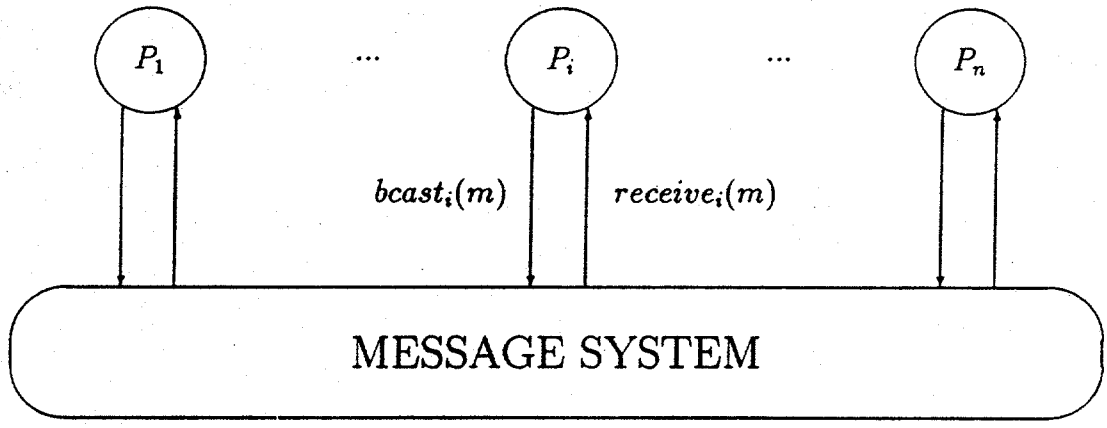


Figure 22.1: I/O Automata Model of the System

that a 1-RCP is necessarily also a 0-RCP.

Definition A *configuration* consists of the set of all process states together with the state of the message system.

Given a configuration C, we say that schedule β can be *applied* to C (or alternatively, that β is *enabled* at C) iff for each $receive_i(m)$ event occurring in β, either the corresponding $bcast_j(m)$ event occurs earlier in β or a pair (m, i) is present in the multiset of the message system in C.

Note that because of the deterministic restriction placed on the processes, the configuration resulting from the application of a given schedule to a given configuration is uniquely determined.

22.1.3 Impossibility Result

Our goal is to show that a 1-RCP does not exist. We will begin by proving a key fact about the commutativity of certain schedules.

Lemma 22.1 *Given a configuration C and two schedules β_1 and β_2 enabled at C involving disjoint sets of processes*[2]*, the schedules $\beta_1\beta_2$ and $\beta_2\beta_1$ are also enabled at C, and when applied at C both lead to the same configuration D.*

[2]For β_1 and β_2 to involve disjoint sets of processes means that if any step of β_1 is a step of process p then no step of β_2 is a step of p.

Proof: If β_1 and β_2 are enabled at C and involve disjoint sets of processes, a straightforward induction shows that $\beta_1\beta_2$ and $\beta_2\beta_1$ must also be enabled at C.

To see that $\beta_1\beta_2$ and $\beta_2\beta_1$ both lead to the same configuration, consider Figure 22.2.

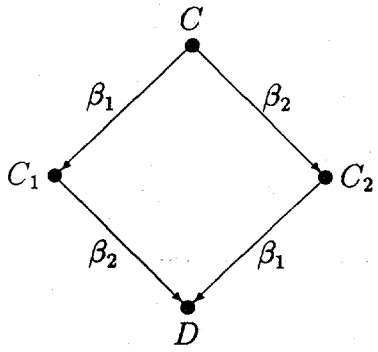


Figure 22.2: Commutativity of Disjoint Schedules

Because β_1 and β_2 involve disjoint sets of processes, the configurations C_1 and C_2 differ from C in a disjoint manner, i.e. the process states that are changed in C_1 are unchanged in C_2 and vice versa, and the elements of the message system multiset that are changed (added/removed) in C_1 are unchanged (absent/still present) in C_2, and vice versa.

We assumed earlier that all process state transitions are deterministic, so applying β_1 to C_2 will result in the same changes to the same set of process states as would occur in applying β_1 to C since C and C_2 agree on the state of all processes involved in β_1.

It follows then that applying β_1 to C_2 produces a configuration which agrees with C_1 on the state of processes affected by β_1 and agrees with C_2 on the state of processes not affected by β_1. Similarly, applying β_2 to C_1 produces a configuration which agrees with C_2 on the state of processes affected by β_2 and with C_1 on the state of those not affected by β_2. In both cases, the resulting configuration agrees with C on the state of processes not affected by either β_1 or β_2. Therefore the configurations resulting from applying $\beta_1\beta_2$ and $\beta_2\beta_1$ to C agree on the state of all processes.

As for message system, the fact that β_1 and β_2 are both enabled at C ensures that there cannot be a $receive_j(m)$ in one schedule for which the corresponding $bcast_i(m)$ is in the other; otherwise one of the schedules would produce an execution which is not well-formed. Therefore, since the message system state changes in C_1 are disjoint from those in C_2, applying $\beta_1\beta_2$ to C produces a configuration with the same message system state as the configuration obtained by applying $\beta_2\beta_1$ to C.

Therefore the configurations resulting from $\beta_1\beta_2$ and $\beta_2\beta_1$ are identical. ∎

During the execution of a given consensus protocol the system proceeds through a sequence of configurations, and at some point it is determined what the decision value of the processes will be. Obviously this must occur by the point where the first process decides, but it may well be that the choice is determined at some earlier point. The following definition clarifies this idea.

Definition A configuration C is *bivalent* if there exist configurations C_1 and C_2, both reachable from C, such that in C_1 some process decides on the value 0 and in C_2 some process decides on the value 1. A configuration is *univalent* if there is only one reachable decision value. A univalent configuration is said to be *0-valent* if the reachable decision value is 0 and *1-valent* if it is 1.

Note that it is not clear at this point that bivalent configurations must exist. In the absence of failures it is easy to see how one could construct a consensus protocol which has a predetermined decision value for each possible set of initial values (e.g., majority). The following lemma shows that the possibility of a fault precludes such a consensus protocol.

Lemma 22.2 *Every 1-RCP has a bivalent initial configuration.*

Proof: Suppose not. Then every initial configuration is univalent. Note that the initial configuration of the system consists simply of the vector of the processes' initial values and an empty multiset of messages. Therefore, each vector in $\{0,1\}^n$ (where n is the number of processes) corresponds to a univalent initial configuration which has some fixed decision value. By the definition of the consensus problem, the vector of all 0's must correspond to a 0-valent configuration, while the vector of all 1's must correspond to a 1-valent configuration.

Now consider the sequence of vectors: 000...0, 000...01, 000...011, ..., 00111...1, 0111...1, 111...1. There must be two adjacent vectors in this sequence (differing in only one element) such that the first corresponds to a 0-valent configuration C_0, and the second to a 1-valent configuration C_1. Let p be the process whose initial value differs in the two vectors.

Consider a 1-fair execution with schedule β in which p takes no locally-controlled steps, leading from configuration C_0 to a configuration where some process q chooses 0 as its decision value. If we now apply β to C_1, the determinism of the processes requires that q must again choose 0 as the decision value, since the difference in p's state is not visible to them and C_1 is identical to C_0 in all other respects. But this contradicts the fact that C_1 is 1-valent. ∎

We now present the main lemma, which claims that in the transition from a bivalent configuration to a univalent configuration there is always some single process which is responsible for the decision, and whose possible failure will prevent the system from reaching a univalent configuration. In the proof of this lemma we use some of the ideas in [BridgelandW87] to give a slightly cleaner argument than is given in [FischerLP85].

Lemma 22.3 *Consider any 0-RCP with a bivalent initial configuration. There exists a reachable bivalent configuration C and a process p such that:*

1. *There exists a schedule involving only steps of p that leads to a 0-valent configuration when applied to C.*
2. *There exists a schedule involving only steps of p that leads to a 1-valent configuration when applied to C.*

The process p is called a decider.

Proof: Suppose not, i.e. assume we have a bivalent initial configuration and that there is no decider. We shall construct a schedule that produces a fair execution when applied from the given configuration, but never reaches a univalent configuration—this implies that the processes never reach a decision, which violates the termination condition for consensus.

Consider a schedule consisting of a sequence of finite rounds such that in each round there is at least one turn for each process, where a turn consists of alternately either letting the process take a locally controlled step, or delivering the oldest message pending for that process (if any). Delivering the oldest message for each process ensures that every message that is sent is eventually received and so is fair to the classes of the message system. Letting each process take another locally controlled step within a finite number of steps ensures that the execution is fair to the processes.

We now need to show that we can construct a schedule with these properties, but that continues to lead to bivalent configurations indefinitely. It suffices to consider a single turn, and then proceed inductively. We shall consider the case of delivering a message to a process during its turn — the case of a process taking a locally controlled step on its turn may be argued similarly.

Assume we have a bivalent configuration C and in the current round of scheduling we need to give p_j a turn to receive a particular message m (the oldest message waiting to be delivered to p_j). We want to find an allowable schedule that delivers m to p_j and results in a bivalent configuration. (We assume there is a message pending for p_j—if not, we don't need to do anything on p_j's turn and we are done.)

Consider the tree of configurations resulting from all possible schedules enabled at C ending in a step in which m is delivered to p_j (i.e. a $receive_j(m)$ action), as in Figure 22.3. If any of the leaves of this tree are bivalent, we may choose the schedule leading to it and we are done. If not, then all of the leaves of the tree are univalent.

We claim that among the leaves of this tree there is a configuration that is 0-valent and a configuration that is 1-valent. We will only prove the existence of a 0-valent leaf—the argument for the 1-valent case is analogous.

The fact that C is bivalent requires that there exist a 0-valent configuration D reachable from C. There are two cases: (1) If message m is delivered to p_j in the schedule β leading

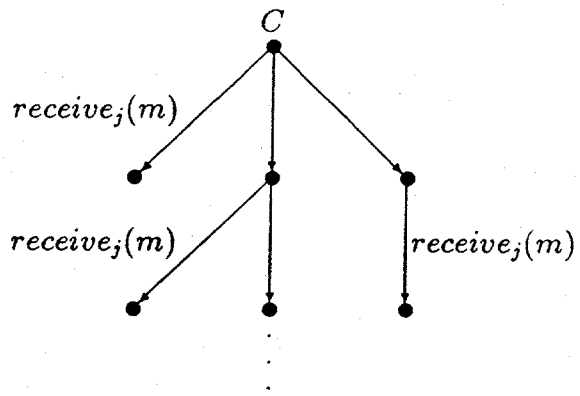


Figure 22.3: Tree of Configurations

to D, let β' be the prefix of β ending in *receive*$_j(m)$ and let D' be the configuration obtained by applying β' to C. Clearly D' is a leaf in our tree, and must therefore be univalent. Since D is 0-valent and reachable from D', it must be the case that D' is also 0-valent. (2) If m is not delivered to p_j in the schedule leading to D then we can append the step of delivering a m to p_j to the schedule leading to D (note that this step must be enabled since it was enabled at C and hasn't yet occurred). The resulting configuration must be 0-valent, since D is, and we have again found a 0-valent leaf of the tree.

The existence of a 0-valent configuration and a 1-valent configuration among the leaves of the tree requires that somewhere in the tree there is an adjacent[3] pair of leaves such that one is a 0-valent configuration (C_0) and the other is a 1-valent configuration (C_1).

The least common ancestor of C_0 and C_1 is a bivalent configuration C'. By the definition of the tree, every leaf is a result of the step *receive*$_j(m)$, and every internal node is the parent of a single leaf, since the tree contains all possibilities. This implies that in order for C_0 and C_1 to be adjacent leaves, C' must be the parent of one of them and the grandparent of the other. Assume without loss of generality that C' is the parent of C_0, and let π be the step leading to the parent of C_1. (See Figure 22.4 for a diagram of this situation.)

Let q be the process involved in the step π. If $q \neq p_j$ then we may apply Lemma 22.1 to show that applying step π to C_0 must lead to C_1 since the steps *receive*$_j(m)$ and π do not involve the same process and therefore commuting them results in the same configuration. But C_0 is 0-valent and C_1 is 1-valent, so we have obtained a contradiction. On the other hand, if $q = p_j$ then p_j is a decider, and this contradicts our assumption that no decider exists.

[3]We say that two leaves are *adjacent* if their parents are connected by an edge in the tree.

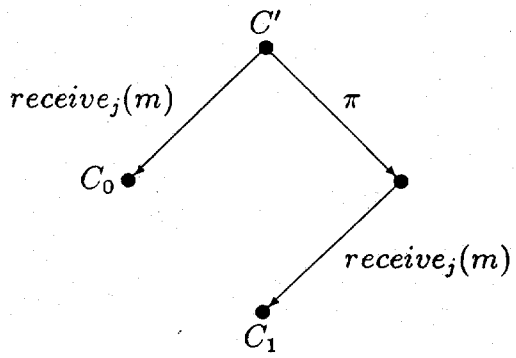


Figure 22.4: Adjacent 0-valent and 1-valent leaves

Therefore the leaves of the tree cannot all be univalent and we are able to construct an allowable schedule which leads to a bivalent configuration. Applying this inductively, we can obtain an execution which never terminates, violating the termination requirement of the consensus problem. ■

We are now ready to prove the impossibility result.

Theorem 22.4 *A 1-RCP does not exist.*

Proof: Assume there exists a 1-RCP. Then by Lemma 22.2 it has a bivalent initial configuration. By Lemma 22.3 there must be a configuration C at which point there is a deciding process p. Now consider an execution α from C in which p takes no locally controlled steps. Assume without loss of generality that in the execution α some process q decides on the value 0. Let β be the schedule associated with α, but with all steps involving message delivery to p removed. In applying the schedule β to C, q cannot tell that p doesn't get its messages, so q must again choose 0.

The process p is a decider, so there exists a schedule γ from C involving only steps of p that leads to a 1-valent configuration. β and γ involve a disjoint set of processes, so by Lemma 22.1 we may apply them in either order and obtain the same configuration. But this is a contradiction since β leads to a 0-valent configuration and γ leads to a 1-valent configuration. ■

22.1.4 Construction of Atomic Test-And-Set Registers

In Lecture 14 it was claimed that it is not possible to construct a wait-free atomic test-and-set register from atomic read-write registers. Using ideas from the above impossibility

result, we are now able to prove this. We use a similar model for the system, except that the message system is replaced by a collection of registers, and the *bcast* and *receive* actions are replaced by invocations and responses for register operations.[4]

We begin by noticing that it *is* possible to solve asynchronous consensus using atomic test-and-set registers. Given an atomic test-and-set register with an initial value of *nil* we can simply have the process that first accesses the register set the register to its initial value and then decide on that value. All other processes will then see that the register has a value other than nil, and decide on that value. Clearly this protocol guarantees agreement and validity, and any process that does not halt will immediately reach a decision, so termination is also satisfied.

Note that this protocol is fully resilient to stop-faults. The failure of any number of processes does not affect the ability of a non-faulty process to access the register.

So, we can solve the asynchronous consensus problem using atomic test-and-set registers. It follows that if the wait-free construction of atomic test-and-set registers from atomic read-write registers is possible, then asynchronous consensus can be solved using atomic read-write registers. Note that the wait-free property of the construction is a key point—if the construction were not wait-free our consensus protocol that used test-and-set registers would not be resilient to halting, since a situation could arise where the register was waiting on a halted process.

We will now show that it is not possible to design a fully resilient consensus protocol using atomic read-write registers. The impossibility of constructing an atomic test-and-set register from atomic read-write registers will then follow immediately from the above argument.

Lemma 22.5 *Given a fully resilient consensus protocol there exists a reachable bivalent configuration C from which every step leads to a univalent configuration. C is called a* deciding *configuration.*

Proof: Suppose that no deciding configuration exists. A fully resilient consensus protocol clearly must tolerate the failure of a single process. Therefore, by a chain argument similar to the one used in Lemma 22.2, we know that there must exist a bivalent initial configuration for the consensus protocol.

Since there is no deciding configuration, from every bivalent configuration we can take some step and get to a new bivalent configuration. By applying this process repeatedly starting at a bivalent initial configuration, we can continue to pass through bivalent configurations indefinitely. It does not matter that the resulting schedule is not necessarily fair, since the consensus protocol is fully resilient,

Therefore in the absence of a deciding configuration we can construct an execution which never leads to a univalent configuration, and this violates the termination condition of the consensus problem. ■

[4]In the proofs, we assume that the invocation-response pair is indivisible.

Now suppose that we have a fully resilient consensus protocol based on atomic read-write registers and consider a deciding configuration C. C is bivalent, so there must be a 0-valent configuration C_0 and a 1-valent configuration C_1 which are reachable in one step. Let π_0 be the step leading to C_0 and let π_1 be the step leading to C_1. (See Figure 22.5.) Let π_0 and π_1 be steps of p and q, respectively.

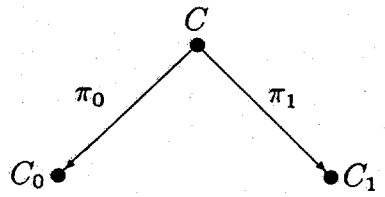


Figure 22.5: Deciding Configuration

Suppose π_0 is a read action for process p. Then consider running q by itself from C_0, starting with π_1. Since C_0 and C are identical except in the local variables of p, it must be the case that π_1 is enabled at C_0, and, moreover, that q eventually decides 1. But this contradicts the fact that C_0 is 0-valent.

Therefore π_0 is not a read action, and similarly, neither is π_1. So they must both be write actions. If π_0 and π_1 write to different registers, then clearly we can apply them in either order and reach the same configuration, but this is a contradiction since C_0 is 0-valent and C_1 is 1-valent.

So π_0 and π_1 must both write to the same register. Consider running q by itself from C_0, starting with π_1. Since C_0 and C are identical in the local variables of q, it must be the case that π_1 is enabled at C_0. And since π_1 overwrites the register written by p in step π_0, q sees the same state that it would if run from C. Therefore, q eventually decides 1, contradicting the fact that C_0 is 0-valent.

We have thus exhausted all possibilities for π_0 and π_1 and must now conclude that a fully resilient consensus protocol is not possible using atomic read-write registers.

22.2 Exercises

1. Prove statements b-e:

(a) If φ is true at all points, then $K_q\varphi$ is true at all points.

(b) If $(\rho,\ell) \models K_q\varphi$ then $(\rho,\ell) \models \varphi$.

(c) If $(\rho,\ell) \models K_q\varphi$ and $(\rho,\ell) \models K_q(\varphi \Rightarrow \psi)$, then $(\rho,\ell) \models K_q\psi$.

(d) If $(\rho,\ell) \models K_q\varphi$ then $(\rho,\ell) \models K_q(K_q\varphi)$.

(e) If $(\rho,\ell) \models \neg K_q\varphi$ then $(\rho,\ell) \models K_q(\neg K_q\varphi)$.

For example, here is the proof of statement a. Suppose $(\rho',\ell') \models \varphi$ for all points (ρ',ℓ'), and consider an arbitrary point (ρ,ℓ). Since $(\rho',\ell') \models \varphi$ for all points (ρ',ℓ'), in particular $(\rho,\ell') \models \varphi$ for all point (ρ',ℓ') satisfying $(\rho,\ell) \sim_q (\rho',\ell')$. It follows that $(\rho,\ell) \models K_q\varphi$.

2. The distributed firing squad problem is defined as follows. An external source may send "start" signals to some of the processors in the system at unpredictable, possibly different, times. It is required that

   (a) if any nonfaulty processor receives a "start" signal, then all nonfaulty processors perform "fire" action at some later point,

   (b) whenever any nonfaulty processor "fires," all nonfaulty processors do so simultaneously, and

   (c) if no processor receives a "start" signal, then no nonfaulty processors "fires."

   This is obviously a synchronization problem: even though processors may receive "start" signals at different times, and in fact it may be that only one of the processors actually receives a "start" signal, we want all nonfaulty processors to "fire" at the same time. When answering the following questions, Handout 25 may be useful.

   (a) Construct a fact φ making the following statement true, and prove that the resulting statement is true: Given a run ρ of a protocol P for the distributed firing squad problem, if the nonfaulty processors "fire" at time ℓ in ρ, then $(\rho,\ell) \models C_{\mathcal{N}}\varphi$.

   (b) State a knowledge-based protocol P for the firing squad problem that is optimal in all runs (similar to the protocol we constructed in class for consensus).

   (c) Prove that the following is true of your protocol P: Let P' be an arbitrary protocol for the firing squad problem. Let ρ and ρ' be corresponding runs of P and P', respectively. If the nonfaulty processors "fire" at time ℓ in ρ', then the nonfaulty processors "fire" no later than time ℓ in ρ.

   (d) Prove that your protocol P solves the firing squad problem and is optimal in all runs. (It may be useful to know that there are protocols for the firing squad problem that halt in $t+1$ rounds in every run.)

3. Prove or disprove: Every 0-resilient consensus protocol has a bivalent initial state.

4. Suppose the network is not entirely asynchronous, in the sense that processes are equipped with perfectly synchronized local clocks, but messages can still take arbitrarily long to be delivered. Show that impossibility still holds for 1-resilient consensus. (Hint: A reduction might be easier than a direct proof.)

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 23: December 6

Lecturer: Nancy Lynch | *Scribe: Atul Shrivastava*

23.1 Randomized Consensus Algorithms

Impossibility results from previous classes require that in a fully connected synchronous network of n nodes with at most t faulty processes, consensus can be reached only if $n \geq 3t+1$. Also, at least $t+1$ rounds are required.

This bound on the number of rounds can be reduced by introducing randomization in consensus protocols. In this lecture, we present Ben-Or's basic idea of introducing randomization to achieve consensus in fewer rounds[Ben-Or83]. Although the first algorithm presented below has exponential expected time, refinements to the algorithm achieve good time performance.

23.1.1 Ben-Or's Randomized Algorithm

The code is shown in Figure 23.1. Each processor starts out with an input bit (x), and agreement, validity and termination conditions are as before. We let *random* denote a coin toss producing 0 or 1 with with equal probability.

We say that one *phase* of the algorithm is the two round synchronous execution by processes. If $n \geq 3t+1$, then Ben-Or's algorithm always preserves the validity and agreement conditions. In addition it also has a high probability of termination. The algorithm is similar to the Turpin-Coan multivalued consensus protocol. In fact, as in Turpin-Coan's algorithm, at most one value (non-nil) is sent by any nonfaulty process during Round 2 in any phase of the algorithm.

Agreement

If any nonfaulty process decides in phase r of an execution, no one else can decide differently at phase r (since nonfaulty processes don't send different values in round 2 of any phase). If this is the case, all nonfaulty processes choose this value for the next phase of the algorithm.

Validity

Suppose all nonfaulty processes start with the same input bit b. In Round 1 of the first phase, all non-faulty processes broadcast b and receive at least $n-t$ messages with value b.

Process p's code: (same for all processes)

repeat forever
Round 1: broadcast x
if $\geq (n-t)$ msgs. recd. with value v
then $x \leftarrow v$
else $x \leftarrow nil$

Round 2: broadcast x
if $\geq (n-t)$ msgs. recd. with value v
then (DECIDE v; $x \leftarrow v$)
else if $\geq (t+1)$ msgs. recd. with value v
then $x \leftarrow v$
else $x \leftarrow random$
endrepeat

Figure 23.1: Ben-Or's Randomized Consensus Algorithm

Then, in Round 2 of the first phase, they will all broadcast b again and receive at least $n-t$ messages with that value. Therefore, all non-faulty processes will decide b (in Round 2 of the first phase). This satisfies the validity requirement.

Termination

In order to send a non-nil value w at Round 2 of any phase, a non-faulty process must have received $n-t$ messages with the same value w in Round 1 of that phase. Therefore, no two non-faulty processes can send different non-nil values in Round 2 of the same phase. So, in Round 2 of any phase, if any two non-faulty process each receive at least $t+1$ (non-nil) messages containing values w_1 and w_2, respectively, then $w_1 = w_2$. Therefore, at most one decision value can be forced on non-faulty processes during Round 2 of any phase of the algorithm.

So, in each phase there are two cases. If a value w is forced in Round 2, then with probability $\geq 1/2^n$ all nonfaulty processes will also choose the value w for the next phase of the algorithm, resulting in termination. If no value is forced in Round 2, then with probability $\geq 1/2^n$ sufficiently many processes will choose the same value to force a value at the next phase. Thus, expected number of phases is at most 2^n.

23.1.2 Improving Expected Time

Rabin suggested that if processes coordinated their coin tosses, with probability $\frac{1}{2}$ all processes will agree on a forced value [Rabin83]. Thus, the expected time is reduced to a constant number of rounds if a global coin tossing mechanism is used. Although it is not clear how this global coin tossing is realized, cryptographic ideas were suggested (e.g., Shamir's secret sharing protocol). Bracha, using Ben-Or's idea, improved the expected number of rounds to $O(\log n)$ assuming private channels for interprocess communication [Bracha87]. Feldman and Micali [Feldman88] realized the global coin tossing idea without using cryptography for a verifiable secret sharing protocol. Their algorithm assumes interprocess communication through private channels. That is, a byzantine process can be seen as an adversary whose behavior is a function of the history of messages arriving on its own channels. Chor and Coan [ChorC87] realized an $O(\frac{n}{\log n})$ bound on expected number of rounds with no cryptographic assumptions.

Thus expected time can be improved using probabilistic algorithms, but the absolute minimum time is still unknown.

23.1.3 Randomized Consensus in Asynchronous Networks

Fischer, Lynch and Paterson show that consensus is impossible in asynchronous environment using deterministic ideas even in presence of stopping faults [FisherLP85]. However, the problem can be solved in asynchronous environment using randomization, with probability 1 of eventually terminating, even in presence of Byzantine faults.

The algorithm, shown in Figure 23.2, is also based on [Ben-Or83]. The algorithm assumes verifiable (but not necessarily secret) message channels. Although the algorithm needs additional processes ($n \geq 7t+1$), this number is reduced to $3t+1$ in [Bracha87], which also uses cryptographic techniques to reduce the expected time from exponential to $O(\log n)$ rounds. The algorithm works in 'phases', where each phase has two rounds. Each process sends messages of the form (r, s, v), where r is the phase number, s is the round number within the phase and v is the 'value' of the message.

The correctness arguments are similar to those for Ben-Or's synchronous algorithm, but the proofs are slightly more complicated in order to deal with asynchrony of phases.

Agreement

Here we show that the nonfaulty processes cannot disagree. Consider the case where p decides v at phase r. This can happen only if p gets $\geq (n-2t)$ occurrences of v, which by counting arguments guarantees that other nonfaulty processes get $\geq (n-4t)$ occurrences of v, hence they cannot decide on a different value in this round. Moreover, agreement will hold for the next phase if v is chosen at the end of the current phase by other nonfaulty processes.

Process p's code:

Initially each process' initial value $= x$
repeat forever

Round 1:
broadcast $(r, 1, x)$
wait for $(n - t)$ msgs. with value $(r, 1, *)$
if $\geq (n - 2t)$ msgs. received with value v
then $x \leftarrow v$
else $x \leftarrow nil$

Round 2:
broadcast $(r, 2, x)$
wait for $(n - t)$ msgs. with value $(r, 2, *)$
Let v = value occurring most often, with $m = \#$ of occurrences
if $m \geq (n - 2t)$
then (DECIDE v; $x \leftarrow v$)
else if $m \geq (n - 4t)$
then $x \leftarrow v$
else $x \leftarrow random$

endrepeat

Figure 23.2: Randomized Consensus Algorithm for Asynchronous Network

Validity

The validity condition is guaranteed using same argument as in the synchronous case.

Termination

The agreement condition shows that if any process decides on v during a phase of the algorithm, termination will occur if all processes choose v at the end of the phase. This event will occur with probability greater than $\frac{1}{2^n}$, same as the result for synchronous case. But for the synchronous case, the forced value was determined by the end of Round 1 during any phase, which is before any nonfaulty process' random choices are made. The same notion should therefore be preserved for this case. Thus, the algorithm must guarantee that no nonfaulty process tosses a coin before this forcible value is determined, to prevent the adversary from using the coin toss results to determine the forcible value. In the next paragraph, we argue that this condition is guaranteed.

Call messages sent in Round 2 *suggestions.* Consider the first process that reaches phase r of the algorithm, at the point where it receives at $(n-t)$ messages of type $(r, 1, *)$. Let v denote the majority value in the messages received (a tie break, if no distinct majority). We then make the following claims.

Claim 23.1 *In phase r, any suggestion sent by a nonfaulty process must have value v.*

Proof: If some other value w was suggested by nonfaulty process p, then p sees $\geq (n-2t)$ messages with value w. Again, by counting arguments, at least $(n-4t)$ messages with value w appear in the set $(n-t)$ processes that suggested w. Since $n \geq 7t+1$, it is clear that $(n-4t)$ is still a majority of nonfaulty processes and thus $w = v$. ∎

Claim 23.2 *In phase r, v is the only possible forcible value.*

Proof: To force a nonfaulty process to choose w, the process must receive at least $(n-4t)$ suggestions of w. At least one of these processes is nonfaulty, so the value suggested is v. Consequently, the forcible value is determined before any nonfaulty process tosses a coin. ∎

Hence, with probability $\geq \frac{1}{2^n}$, all processes tossing coins will choose v, and will then decide v in the next phase.

23.2 Concurrency Control

The last major topic covered in the class is Concurrency Control. During this course, many distributed algorithms and impossibility results in the field were discussed. We now consider principles for programming distributed systems. The languages used in programming distributed systems vary widely with the application (communication protocols, real-time

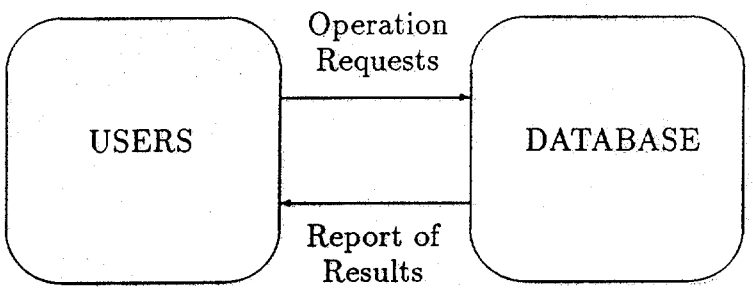


Figure 23.3: Atomic View of a database system

process control systems, distributed data-processing systems, resource management systems etc.). A lot of research has been done in design of languages for distributed data processing systems.

The remaining lectures will be devoted to study of important language constructs for data-oriented distributed programming. The text for the material is the monograph by Lynch, Merritt, Weihl and Fekete titled *Atomic Transactions*. We begin by introducing basic distributed data processing concepts, covered in Chapters 1 and 2 of the monograph.

The operational view of a database system is shown in Figure 23.3. This view is similar to an atomic register, in that an operation is performed on the database as if it was shrunk to a point. Consequently, the operations appear to be performed in a serial fashion.

But the analogy to the atomic register stops here. In the case of atomic transactions, some operations may not succeed. Successful operations COMMIT, whereas operations that do not succeed ABORT. As a result, the wait free property in atomic registers is also not preserved by transactions.

Since transactions may or may not succeed, this may result in two different courses of action for the creator of the transaction. Thus the result of each transaction invoked is reported back to its creator. Report for COMMIT transactions is accomplished by a REPORT_COMMIT(T,v), where T denotes the transaction name and v is the value returned by the successful completion. For transactions that ABORT, the results are reported using REPORT_ABORT(T) action. For obvious reasons, a REPORT_ABORT does not have a return value v.

A sequence of operations within an atomic transaction (considering COMMIT and ignoring ABORT operations) can be viewed as a serial execution.

In a typical database, the system behavior can be partitioned into separate *objects*. Hence, an operation may involve several objects. As a result, an operation can be subdivided into operations on individual objects. For example, a bank transfer operation TRANSFER(A,B,k) first withdraws $k from Account A and then deposits $k into Account B. Consequently, the TRANSFER(A,B,k) can be decomposed into WITHDRAW(A,k) followed by DEPOSIT(B,k) operations.

Thus a transaction can be thought of as a sequence of operations, involving many objects. But the semantics of the transaction should be such that it appears to the outside world that the whole transaction happened atomically.

Example: Consider the bank AUDIT operation, which returns the sum of money in all accounts. The AUDIT operation could be realized in a logical way by expanding it into a transaction that reads values of balances in all accounts one at a time, in some order. If operations could interleave other transactions say TRANSFER(A,B,k) as before, the AUDIT could end up returning an incorrect total. Thus a correct operation for the banking system should make it appear to the user as if the AUDIT and TRANSFER operations happen consecutively.

In addition, a transaction could also include conditionals, which may result in different actions. In the banking system example, a WITHDRAW(A,k) transaction may take a different action if it finds an insufficient starting balance. Thus we use the word transaction to describe a 'program'. In our case, we choose to model transactions as I/O automata.

A transaction T is shown in Figure 23.4. T wakes up when it gets a special create input, and starts computing. In turn T may request that certain operations (its children) be invoked and their results be reported back. The children in turn get created and presumably operate directly on the data. A REPORT_COMMIT with results or a REPORT_ABORT reports the outcome of each child transaction. The transaction may choose to use information about ABORT as well as COMMIT operations of its children to decide its own course of action. Eventually T completes and requests to commit with its own result v. All these actions should appear atomic from outside of T.

Consider the actions where T invokes its children. In classical database theory, these actions occur sequentially, waiting for report of one before requesting the next, to ensure that they occur in a given order. In general, this may not be necessary. Thus, T could take actions invoking its children concurrently. Again, this results in semantics that are similar to an atomic register. If T requires that its children be executed in a certain order, then it would wait for a report about each child before invoking the next transaction.

Readers familiar with the classical approach of modeling transactions (covered in the book *Concurrency Control and Recovery in Database systems* by Bernstein, Hadzilacos and Goodman) will recall that transactions are viewed as a two level structure. Instead of stopping at the second level, we shall model transactions as nested structures with no restrictions

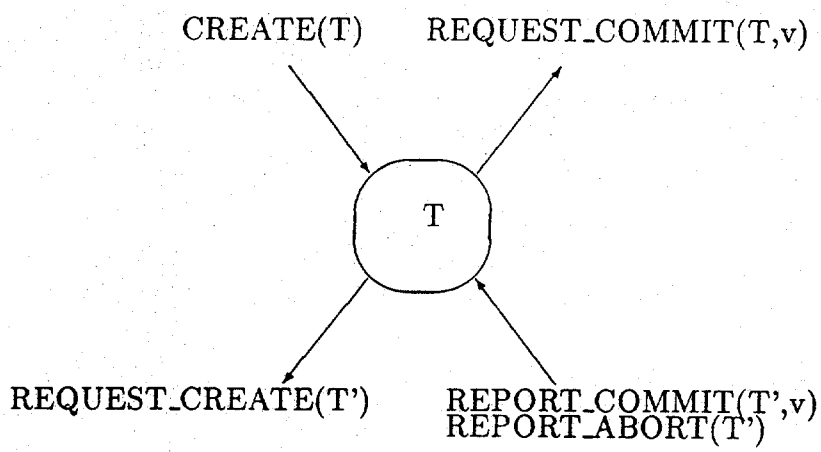


Figure 23.4: Transaction Model

on the level of nesting. Operations can thus be expanded into transactions at a next level. Semantics say that each set of siblings execute in accordance with the shrinking definition - as if serially and consistent with the order of invocations and responses.

Nested transactions are convenient structures to use as a basis for distributed programming languages such as the Swallow project, Argus, Camelot, etc.

Consider a distributed system in which a centralized serial scheduler gets requests from all transactions and runs them one at a time. This system has no concurrency and is not of much practical interest. On the other hand, such a system can be used as the basis of a correctness definition. A scheduler that allows concurrency is correct as long as it preserves the properties exhibited in some serial execution of the system.

This can be achieved by using exclusion mechanisms such as locks. A transaction goes around locking all objects that it modifies. These locks are retained until the transaction completes its operation. Thus no two operations on the same object can appear interleaved. Mechanisms such as timestamps and version management can be used instead of locking to achieve the same effect.

Consider the case where a transaction makes a lot of modifications to the database and is then aborted for some reason. Here, mechanisms should ensure that no effects of the aborted transaction show up later. Thus, *recovery* is also an important part of transaction algorithms. Recovery mechanisms also include handling database consistency problems occurring due to

crashes in a system which result in loss of information.

We will be concerned only with safety properties, and will not deal with liveness issues.

6.852 Distributed Algorithms Fall Semester, 1988

Lecture 24: December 8

Lecturer: Nancy Lynch *Scribe: Sanjay Ghemawat*

Atomic transactions were introduced in Lecture 23 as means of handling concurrency and limiting the effects of failures in a distributed data processing system. We now present formal notions of what it means for a transaction system to be "correct". Most of the terms used in the following presentation are not defined here. *Atomic Transactions* (by Lynch, Merritt, Weihl, and Fekete) should be consulted for the appropriate definitions. The reader will be referred to the book as necessary in the following discussion.

24.1 I/O Automata

We model the individual components of a transaction system with I/O Automata, and model the whole transaction system with the composition of the individual I/O Automata. The rest of this section is just an intuitive description of some of the key ideas from Chapter 3.

24.1.1 Modification to the I/O Automaton Model

The original I/O automaton model included a partitioning of locally controlled actions to model fair scheduling of different threads of control in an automaton. This was used to prove fairness and liveness properties about the composition of I/O automata. Since we will only be considering safety properties in the following discussion, partitioning of locally controlled actions is not needed.

24.1.2 Implementations

Correctness proofs presented in later lectures will prove the correctness of very general non-deterministic algorithms. We would like to be able to reuse these proofs for less general, more deterministic versions of these algorithms. Therefore, we need to rigorously define what it means for an automaton A to *implement* another automaton B. If we can prove that A implements B, and we can prove some properties about B, then the same proofs can easily be extended to A.

24.1.3 Possibilities Mappings

An easy way to show one automaton implements another is by demonstrating a correspondence between the states of the two automata. Such a correspondence is called a *possibilities*

mapping. Refer to the book for a formal definition of a possibilities mapping; and a proof that if a possibilities mapping from automaton A to automaton B exists, then A implements B.

24.1.4 Safety Properties

Since automata in this model are unable to block input actions, it is often convenient to restrict attention to those behaviors in which the environment provides "sensible" inputs to the automaton, i.e., the environment obeys certain "well-formedness" restrictions.

A useful way of discussing such restrictions is by saying that an automaton "preserves" a property of behaviors of the system: i.e., as long as the environment does not violate the property, neither does the automaton. Such a notion is primarily interesting for safety properties. A *safety property* is a predicate P on the set of behaviors of the system such that if P holds for a behavior β of the system, then P holds for all prefixes of β. Refer to the book for a formal definition of when an automaton preserves a safety property.

24.2 Transaction System Model

We now present an intuitive description of the system model used in the following discussion. For a formal presentation of the model, refer to Chapter 4 (Serial Systems and Correctness) of the book. The system contains several transactions and objects. A transaction is allowed to access the objects, or invoke other transactions, or both. This nesting of transactions gives rise to a forest of transactions with top-level transactions at the roots of the forest, and accesses to objects at the leaves of the individual trees. To simplify the discussion, we convert this forest to a tree structure by adding a "dummy transaction" as the root of the transaction structure, and by making all the top-level transactions children of this root transaction. This root transaction can be thought of as modeling the outside world from which invocations of top-level transactions originate, and to which reports of the results of such transactions are sent. We will generally regard the boundary between this root transaction and the rest of the system as the user interface to the system.

24.2.1 Serial System Model

We now model a *serial transaction system.* A serial transaction system model consists of transactions as in Figure 24.1, a set of serial object automata, and a serial scheduler automaton (see Figure 24.2). A serial system runs sibling transactions sequentially, and aborts a transaction only if the transaction has not actually been "started". Therefore, a serial system looks like a data processing system in which execution is sequential, and no activity needs to be undone.

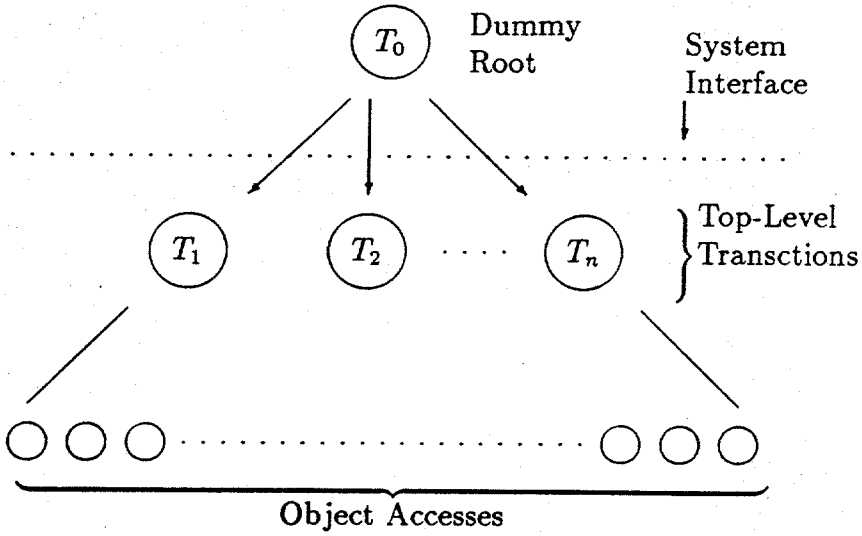


Figure 24.1: Transacation System Model

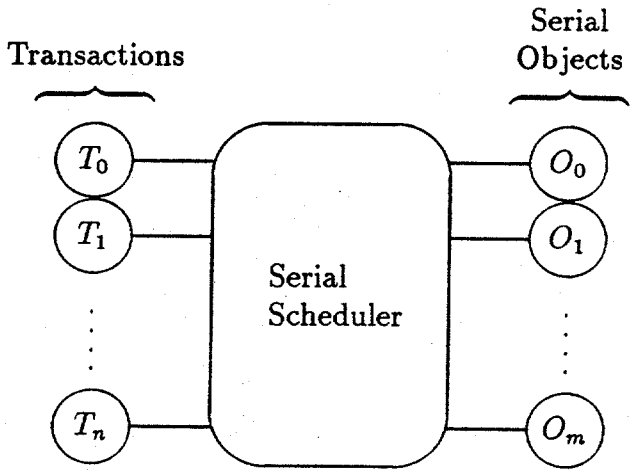


Figure 24.2: Serial System Model

Transaction Automaton

A transaction automaton T has the signature denoted by Figure 24.3.

CREATE(T)	Tells T to start itself.
REQUEST-COMMIT(T,v)	Asks scheduler to commit T with return value v.
REQUEST-CREATE(T')	Asks scheduler to create child T'.
REPORT-COMMIT(T',v)	Reports commit of child T'.
REPORT-ABORT(T')	Reports abort of child T'.

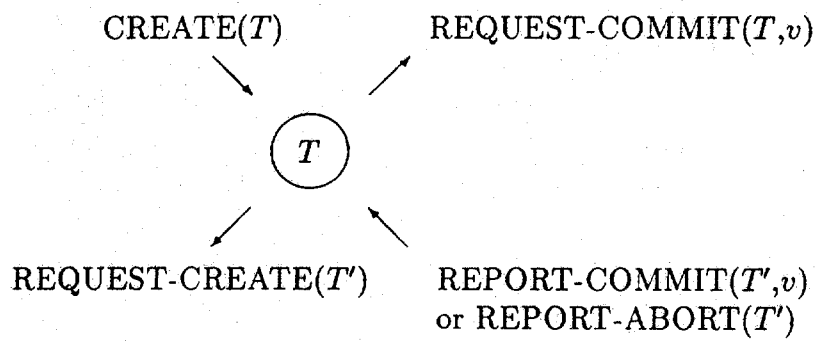


Figure 24.3: Transaction Automaton

Note that there is no REQUEST-ABORT(T) in the signature. This is because the scheduler, since it is non-deterministic, could decide to abort T on its own. Therefore, a REQUEST-ABORT operation is not needed.

Serial Object Automaton

A serial object X serially executes the operations invoked on it. Object X has an input action CREATE(T), and an output action REQUEST-COMMIT(T,v) for each access transaction T for X (see Figure 24.4).

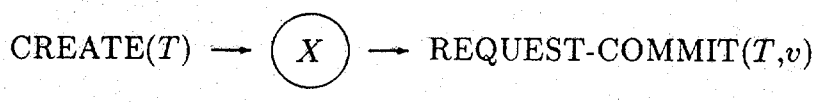


Figure 24.4: Serial Object Automaton

Serial Scheduler Automaton

A serial scheduler accepts requests to create and to commit transactions. It decides when to actually create and commit the transaction. It might also decide to abort a transaction T provided that T has not yet been actually created. The interface of a serial scheduler is summarized in Figure 24.5. Note that the COMMIT and ABORT actions appear as outputs for technical convenience. They do not compose with any input and can be thought of as internal actions.

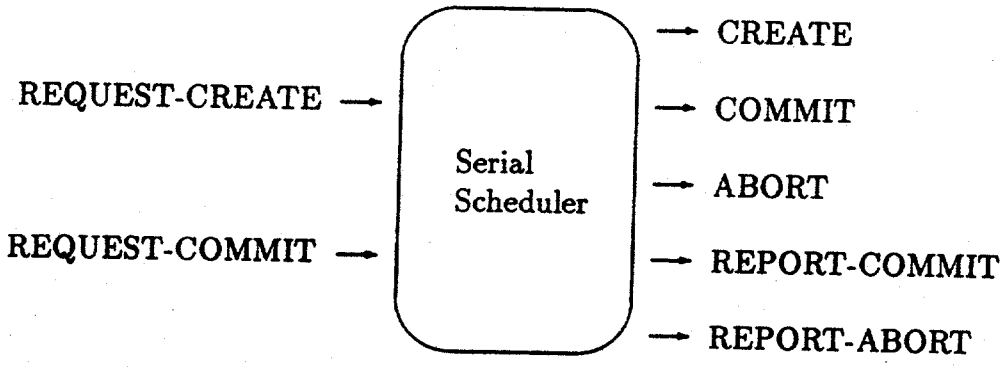


Figure 24.5: Serial Scheduler Automaton

24.3 Well-Formedness

We now describe some properties that will be preserved by the components of the serial system.

24.3.1 Transaction Well-Formedness

1. Transactions must be created before anything else happens at them.
2. A transaction may be invoked (created) at most once.
3. A transaction can be committed or aborted at most once, and cannot be aborted if it is created.
4. A transaction can be committed or aborted only after its creation has been requested.
5. A commit can be requested for a transaction only after all created children have been either committed or aborted.

6. Nothing happens at a transaction after it requests to be committed.

24.3.2 Serial Object Well-Formedness

Operations on an object must be invoked serially; i.e., an operation on an object must be allowed to finish (REQUEST-COMMIT) before another operation (CREATE) is invoked on it.

24.4 Correctness Conditions

Intuitively, we say that a transaction processing system is correct if it looks like a serial system to the world. We define a sequence β of actions to be *serially correct* for a transaction T provided there exists a behavior γ of a serial system such that $\beta|T = \gamma|T$. Given this definition, a transaction system A is *serially correct* for transaction T if all of its finite behaviors are serially correct for T. This means that at transaction T, any execution of system A looks like an execution of a serial system.

We use serial correctness for T_0 (the dummy root transaction) to check whether a transaction system appears correct to the external world. Note that even though A might appear serially correct to the external world, internal components of A might witness non-serial behavior. Refer to the book for some stronger correctness conditions which handle this.

24.5 Exercises

1. Design a serial system representing a banking system. The system should contain a serial object corresponding to each of a fixed finite number of bank accounts, and should have top-level transactions of types WITHDRAW(A,k), DEPOSIT(A,k), TRANSFER(A,B,k) and AUDIT. (There may also be subtransactions.) The WITHDRAW and TRANSFER transactions should have no effect if there are insufficient funds in the account.

   Describe the transactions and serial objects of the system. Give the nesting structure, and give specific I/O automata for the serial objects and transactions. Be sure to include descriptions of what a transaction does when the abort of a child transaction is reported to it.

2. Consider a subsystem S of your system in 1. above, another serial system consisting of TRANSFER and AUDIT transactions only (but no WITHDRAW or DEPOSIT transactions). Suppose that the total of all the money initially in the accounts is k dollars.

(a) Sketch a proof that the answer returned by any AUDIT transaction in S is k.

(b) Suppose another system S' is serially correct for T_0, with respect to the serial system S. Prove that the answer returned by any AUDIT transaction in S' is also k.

3. Prove the lemma about the elementary properties of "visibility" (in Chapter 6 — The Serializability Theorem).

4. Consider the lemma in Chapter 6 which states that all members of pictures(β,T,R) are finite behaviors of the serial scheduler. The definitions of "pictures" given in Chapter 6 requires that γ be chosen to be a certain prefix of δ, rather than all of δ. Why is this "chopping" needed in the proof of the lemma?

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 25: December 13

Lecturer: Nancy Lynch | *Scribe: Jon Riecke*

25.1 Serial Correctness

In the last lecture, we defined a *serial system* for processing transactions. Basically, the serial scheduler proceeds depth-first through the transaction tree, scheduling transactions one at a time. Such a system yields no concurrency, and so has little practical import; the purpose of the serial system is to formalize the correctness of an arbitrary concurrency control algorithm. The serial system is analogous to a serial register: as the correctness of an atomic register implementation can be stated in terms of the behavior of a serial register (*i.e.*, where reads and writes are processed sequentially), so can the correctness of a concurrency control algorithm be stated in terms of the serial system.

Suppose we are given an arbitrary concurrency control algorithm; from the perspective of the user, top-level transactions are invoked and results appear via REPORT actions. In our model, the transaction T_0 represents the user (see Figure 25.1). In order for the

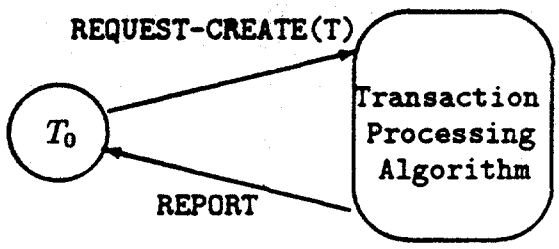


Figure 25.1: A transaction processing system, from the perspective of the user.

implementation to be correct, results returned by the implementation must look the same as some execution of the serial system. More formally, recall from Lecture 24 that an implementation is *serially correct for* T_0 if, for all finite behaviors β, there is a behavior γ of the serial system with $\beta|T_0 = \gamma|T_0$.

In a serially correct transaction processing algorithm, one can imagine each top-level (*i.e.* user) transaction as being *atomic*. An interval corresponding to a transaction consists of five actions: REQUEST-CREATE(T), CREATE(T), REQUEST-COMMIT(T), COMMIT(T), and a REPORT-COMMIT(T,v) (or ABORT replacing COMMIT), where the middle three

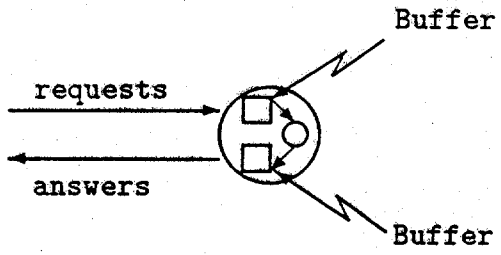


Figure 25.2: An atomic register implemented along the lines of a transaction processing system.

actions are invisible to the user. The user may see a delay between the REQUEST-CREATE and actual CREATE, and between the COMMIT and actual REPORT-COMMIT actions. Nevertheless, due to the serial correctness of the system, one may think of these actions as happening at a single instant—an interval shrinking argument, as is used in the case of atomic registers.

It is interesting to note that the correctness of an atomic register may be formulated along analogous lines. Any atomic register may be thought of as having two buffers: one to receive read and write actions, and one to store return results and acknowledgments (see Figure 25.2). Reads or writes come into the register, the register processes them with some delay, the register reports its results, and the user finally receives the results after some delay.

25.2 A Note of Comparison

By way of comparison, there are at least three differences between this model and the model described in Bernstein, Hadzilacos and Goodman's book [BernsteinHG87]. First, the model here incorporates nested transactions—a feature of many distributed database systems, including Argus. Second, the model incorporates an interface to the user, the transaction T_0, which coordinates all other transactions. This extra top-level transaction greatly simplifies the theory by making the model more uniform. Finally, the model is more detailed than the model in [BernsteinHG87]. Of course, proofs may become hairy using the more detailed model, but the model is more flexible and so more concurrency control algorithms can be modeled using the framework set forth.

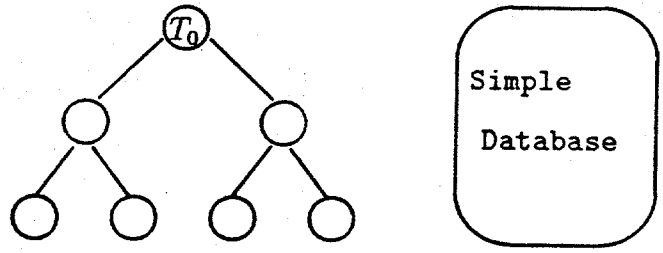


Figure 25.3: The simple transaction system.

25.3 Serializability Theorem

Verifying the serial correctness of a transaction processing algorithm can be difficult given only the definition of serial correctness: we must be able to construct a serial schedule for each schedule of the implementation. Not only is the effort potentially great, the argument must be duplicated for each implementation we wish to verify. Given the large number of concurrency control algorithms in existence (*e.g.*, locking, time-stamp algorithms), the task seems daunting.

Fortunately, the proofs of most transaction processing algorithms can be based on a single general theorem, the *Serializability Theorem*: if certain local conditions on the transactions, objects, and scheduler are met, and some global ordering on the transactions is known (which may be partial), then there is a behavior of the *serial* system that produces the same results from the perspective of T_0. In other words, the Serializability Theorem shows that the system will be serially correct.

The Serializability Theorem is proved for a very general transaction processing system, which we call the *simple system*; there will be a correspondence (via a possibilities mapping, for example) between the simple system and most transaction processing algorithms, as we shall see in Lecture 26. The simple system is composed of a transaction tree and a *simple database*, which both schedules the transactions and models the objects (see Figure 25.3.) A complete I/O automaton definition of the simple database is given in *Atomic Transactions* [LynchMWF88], Chapter 6, pages 65–67. The simple database is basically the minimum system that produces well-formed behaviors (*e.g.*, responses follow the appropriate requests.) If an algorithm implements the simple system (*i.e.*, its behaviors are a subset of the simple system's), then the transactions, objects, and scheduler will produce well-formed behaviors; this is the local condition mentioned above.

The simple system will not, however, always produce serially correct behaviors (this is a necessity if we wish to use the simple database to prove the correctness of a wide variety of algorithms.) One way to derive a non-serially correct behavior for a transaction T is to

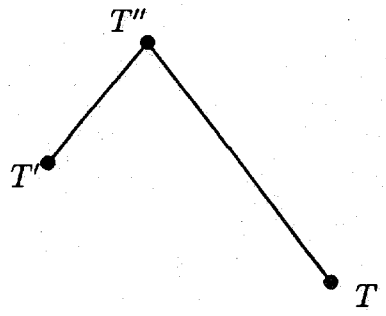


Figure 25.4: A piece of the transaction tree, with T'' the least common ancestor of T and T'.

access an object based on information that could not be known to T. For example, suppose we have the subtree depicted in Figure 25.4 in our transaction tree. T should not be able to use any information about the results of T' unless T' has COMMITted; then the information could be "passed" to T via T'' during the CREATE(T). A transaction T' is thus said to be *visible* to T in a behavior β if there is a COMMIT(U) action for every ancestor U of T' up to the least common ancestor of T and T' (see Figure 25.5.)

The sequence of actions visible to T in β is denoted visible(β, T); a precise definition may be found in Chapter 6 of [LynchMWF88]. Since a transaction T' is either visible or not visible to T in any particular behavior β, all of the actions or none of the actions of T' are in visible(β, T); this important property is formalized on page 71 of [LynchMWF88], in Lemma 6-9.

In a serially correct behavior, transactions are run in an order where all child transactions of a particular transaction must complete before a sibling starts. We call an irreflexive partial order R on a transaction tree a *sibling order* if R relates only siblings.[1] For example, in Figure 25.5, T cannot be related to T' nor T'' via a sibling order. In order to extend a sibling order R to order non-siblings, let R_{trans} be an extension of R such that $R_{trans}(T, T')$ when T is a descendant of U, T' is a descendant of U' and $R(U, U')$. Finally, let $R_{event}(\beta)$ (where β is a behavior) be a sequence with the events ordered by R_{trans}; see Chapter 6 of [LynchMWF88], page 74 for a precise definition.

Not all sibling orders are useful; we must, for example, make sure that all siblings are ordered if we wish to obtain a serial-like behavior. We call a "good" sibling order *suitable.* More precisely, a sibling order R is *suitable* for a behavior β and transaction T if

[1] The partial order must be irreflexive to avoid identifying transactions. For example, suppose R were a partial order with $R(T, T')$ and $R(T', T)$; by the antisymmetry axiom, $T' = T$, something we do not wish to permit.

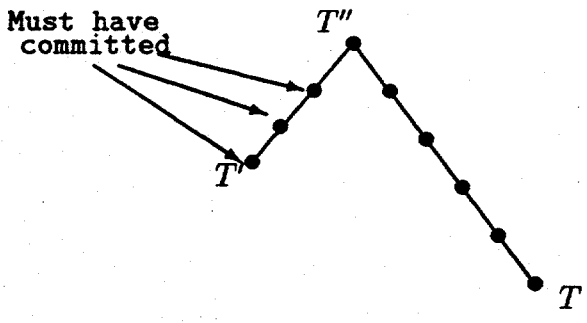


Figure 25.5: Some transactions that are visible to T—those that may affect T in a serially correct behavior.

- R orders all pairs of siblings in **visible**(β, T); and
- R_{event} and **affects**(β) are consistent partial orders on the actions in **visible**(β, T).

Here, **affects**(β) is another irreflexive partial order that orders the actions in β so that a reordering of β consistent with **affects**(β) is a behavior of the simple system. A complete definition of **affects**(β) may also be found in Chapter 6 of [LynchMWF88].

These are the definitions we need in order to prove the Serializability Theorem, which we now state:

Theorem 25.1 (Serializability) *Let β be a finite behavior of the simple system, T a transaction name such that T is not an orphan in β, and R a sibling order suitable for β and T. Suppose that for each object name X, the actions in* **visible**(β, T) *occurring at X and reordered by R, denoted* **view**(β, T, R, X)*, is a finite behavior of the* serial *object S_X. Then β is serially correct for T.*

Both a rough sketch and a complete proof of this important theorem appear in Chapter 6 of [LynchMWF88], pages 76–81.

In Lecture 26, we shall see how to apply this important theorem to proofs of correctness for various concurrency control algorithms.

6.852 Distributed Algorithms | Fall Semester, 1988

Lecture 26: December 15

Lecturers: Nancy Lynch, Ken Goldman | *Scribe: George Varghese*

This lecture has two major parts. The first part describes how to apply the Serializability Theorem to provide a general proof frameork for Locking and Timestamp based concurrency control algorithms. The second part describes a modular proof technique for proving the correctness of data replication algorithms.

26.1 Locking Algorithms

Chapter 7 of [LynchMWF88] describes a proof technique for locking algorithms in nested transaction systems. We will not repeat this description here; instead, we only make a few comments and refer the reader to specific portions of Chapter 7. We will also refer the reader to portions of preceding chapters for an explanation of necessary terminology and sometimes to provide a contrast.

In the previous lecture, we studied the Serializability Theorem, as described in Chapter 6 of [LynchMWF88]. Essentially, to apply this theorem to show that a concurrency control algorithm is correct, we must

- find a suitable sibling order R, and
- show that every object in the system satisfies the view condition required in the hypothesis of the Serializability Theorem.

This technique is applied to Locking Algorithms in Chapter 7 of [LynchMWF88]. For locking algorithms, the sibling order we choose is simply the completion order of transactions. Intuitively, this is because concurrent transactions are not given conflicting locks until the transactions holding them commit. Thus transactions with conflicting accesses to objects are serialized by the order in which they obtain conflicting locks. But this order can be inferred (by the preceding argument) by looking at the order in which they commit, since locks are held until commit. Chapter 7 formalizes these ideas.

In Chapter 7, locking algorithms are modelled using a very simple and unrestrictive controller (called a generic controller). The concurrency control algorithms are modelled as residing in the objects themselves. Such objects are called generic objects. The whole system model (i.e. the composition of the transaction automata, generic object automata, and generic controller) is called a generic system (Section 7.2.2). This is illustrated in Figure 26:1.

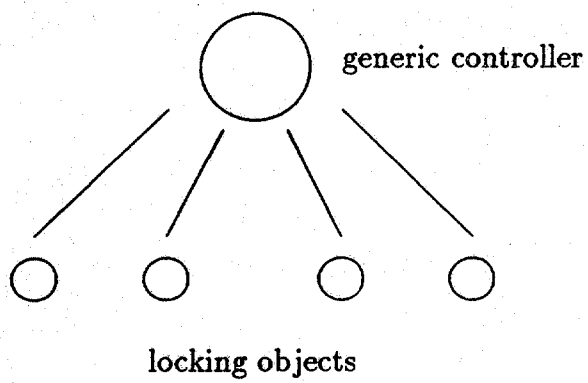


Figure 26.1: Generic System: a model for locking algorithms

If objects are to do all the concurrency control work in this model, they need to know the fate of some of the transactions, especially those that access objects. To model this, we add two input actions to the usual object automaton signatures (compare with serial object automaton signatures): INFORM_COMMIT and INFORM_ABORT can be used by the generic controller to tell objects about the fate of arbitrary transactions. This is described in Section 7.2.2.1. It is also useful to compare the well-formedness conditions for generic objects with the corresponding conditions in Section 4.4.2.2 for serial objects.

The generic controller automaton is described in Section 7.2.2.2. It is very useful to compare this to the corresponding automation for the serial system controller (Section 4.5.3) and the simple database (Section 6.1.1). Unlike the serial scheduler, the generic controller does not restrict concurrency in any way. It is much more like the simple database; both of them only try to preserve some basic well-formedness conditions and impose no other restrictions on concurrency. The transition relation for the generic controller is identical to that of the simple database of Section 4.5.3 except for the additional input events for INFORM_COMMIT and INFORM_ABORT.

To apply the Serializability Theorem, we need to show that a certain view condition holds for all objects in the system, assuming the use of completion order as a suitable sibling order. This gives the proofs a nice modularity because we can show this condition (which is called dynamic atomicity, Sec 7.2.3) separately for each object. However, the view condition is

defined (Section 6.4) in terms of the system containing the object, and not the object itself. Consequently Section 7.2.4 introduces a notion of local dynamic atomicity, a condition that is expressed only in terms of the behaviour of the object under consideration. Section 7.2.4 shows that local dynamic atomicity is sufficient to show dynamic atomicity. Showing that each locking object separately satisfies local dynamic atomicity makes the proofs even more modular.

Section 7.3 applies this modular proof technique to a commutativity based locking algorithm. This commutativity based locking algorithm generalizes several existing locking algorithms. A simpler proof of correctness (based on invariants) has since been found and will probably be added to a later version of [LynchMWF88].

Finally Section 7.4 presents Moss's algorithm for read-update locking and its proof. Moss's algorithm is shown to be a special case of the commutativity based locking algorithm, except that it uses a more efficient data structure to summarize lists of operations at the cost of reduced concurrency. Because the data structures are very different, the Moss objects (M_X) cannot be treated as a special case of a commutativity based locking object (L_X). Instead Section 7.4.2 described a possibilities mapping between M_X and L_X; But Section 7.3.2 has already shown that L_X is dynamic atomic; thus M_X is also dynamic atomic. The serializability theorem then allows us to conclude that Moss's algorithm is correct.

26.2 Timestamp Algorithms

Chapter 8 of [LynchMWF88] provides a framework for proving the correctness of timestamp based concurrency control and recovery algorithms for nested transactions.

The technique that was applied to locking algorithms in Chapter 7 of [LynchMWF88] is applied to timestamp based algorithms. A model of a general system that uses timestamp based concurrency control is shown in Figure 26.2. The generic controller of Chapter 7 has been replaced by a pesudotime controller and the locking objects by pseudtotime objects.

The basic notion is that before each transaction gets created a pseudotime interval (Section 8.2.1) is assigned to it. Different top-level transactions are assigned disjoint intervals. However, subtranactions are assigned time intervals within the parent's interval.

The treatment is very similar to that of locking algorithms; we refer the reader to Chapter 8 for details. There are some interesting differences to note in the two frameworks. First, for timestamp based algorithms, the sibling order we choose is the pseduotime order (Section 8.2.2.2) and not completion order. Second the pseudotime objects (Section 8.2.3.1) need to be informed about the timestamps of transactions as well as their fates; locking objects need to know only about the fate of transactions. Third, to apply the serialability theorem, we can test each pseudotime object separately for a condition called local static atomicity (Section 8.2.4 and 8.2.5); the equivalent condition for locking objects is local dynamic atomcity. Fourth, in Chapter 8 considerable stress is laid on distributed implementations and their

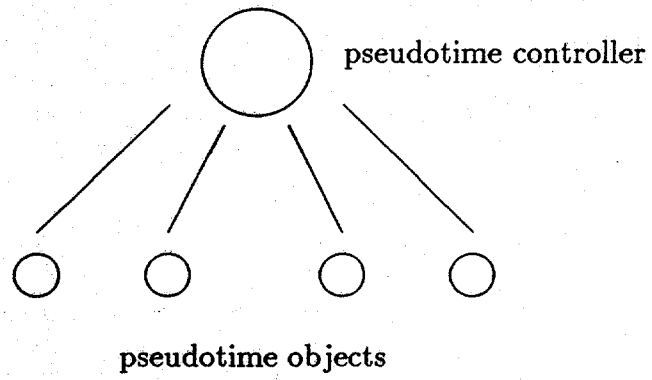


Figure 26.2: System model for Timestamp based locking algorithms

proofs.

Using this framework, Section 8.3.1 proves that a generalization of Reed's algorithm [Reed78] is correct. Section 8.3.3. models Reed's original algorithm and proves that it is correct by describing a possibility mapping between it and the more general algorithm of Section 8.3.1.

26.3 Data Replication Algorithms

We now describe a framework for proving the correctness of data replication algorithms in nested transaction systems.

Data replication is the technique by which we keep more than one copy of each data item at different nodes in the network. The advantages of data replication are:

- Database Availability: The database can continue to make progress even if some nodes in the network crash.
- Reliability: Data can survive a catastrophe at some of the machines.
- Performance: By storing copies of the data close to the points at which they are accessed, we can reduce access time.

These advantages are, however, purchased at a price. We must maintain some kind of consistency among the multiple copies of the same datum. Achieving this consistency is the purpose of a data replication algorithm. There are several such algorithms in the literature. Here we focus on a framework developed in [GoldmanL87] for proving the correctness of data replication algorithms in the context of nested transaction systems.

It happens that we can do nearly all our reasoning in terms of Serial Systems. This gives the proofs a very nice modularity: we can concentrate on proving that the replication algorithms have the desired properties independent of the concurrency control and recovery, which can be proved separately.

To capture the essential features of items in the database, we use a *logical data item* X (V_x, i_X), where V_X is the domain of values for X and i_X is the initial value of X.

To model the replicas we use *read-write objects*, which have two kinds of accesses: write accesses (that have an associated value), and read accesses (that return the value associated with the preceding write access).

26.4 Quorum Consensus Algorithm

We consider the Quorum Consensus algorithm of Gifford as an example of a data replication algorithm. We will use this to introduce (and test) a correcteness proof framework for data replication algorithms.

26.4.1 Description

We briefly describe the basic quorum consensus algorithm [Gifford79].

For each logical data item X, we associate some number n of replicas. Each replica is modelled as read-write objects $X_1, X_2, ...X_n$, each with initial value $(i_X, 0)$, where the first element in the tuple is the initial value and the second the initial version number.

Associated with X is a piece of state called the *configuration*. A configuration consists of a set of *read quorums* and a set of *write quorums*. Each read or write quorum is a set of names of replicas such that every read-quorum has a non-empty intersection with every write-quorum (quorum intersection rule).

The algorithm that follows is reminiscent of some of the atomic register constructions we have seen (i.e. logical operations decompose into little sub-operations).

The algorithm to do a $READ_X$ is as follows. First do a read access at all replicas in some read-quorum for X. Return the value at the replica with highest version-number.

The algorithm to do a $WRITE_X$ is as follows. First do a read access at all replicas in some read quorum of X. Let $NewVn$ be the maximum version number read plus 1. Finally do a write access with $(v, NewVn)$ at all replicas in some write quorum for X.

This algorithm generalizes the read-one/write-all and read-majority/write majority schemes.

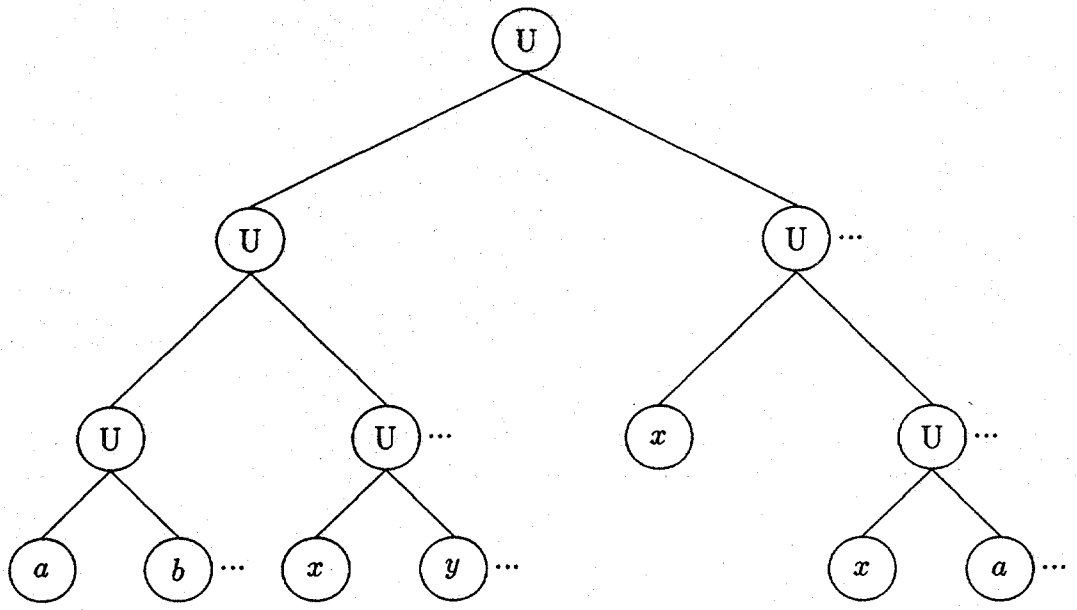


Figure 26.3: System A

We begin considering the proof of this algorithm assuming that the configuration is fixed. Later we consider a modification that allows *reconfiguration.*

26.4.2 Proof for a fixed configuration

We consider three systems for the purposes of the proof. Figure 26.3 depicts a non-replicated serial systems which we will call System A. To do a read we simply do a read acess, and to do a write we do a write access.

Figure 26.4 depicts a replicated but serial system that we call System B. System B has two kinds of Transaction Managers (TMs): Read TMs and Write TMs. The children of the TMs are accesses to the replicas. However, to the users (U's in the figure), the TM's interface looks like the interface to the corresponding read/write objects in System A.

Finally, we consider a third system we call system C which looks like B but allows concurrency. For example, this can be done by replacing the read/write objects with locking objects and the serial scheduler with a concurrent scheduler. Clearly System C is the system of interest.

We now give a high level, top-down description of the proof. See [GoldmanL87] for more details.

First we need to show that the serial but replicated system B simulates the serial and

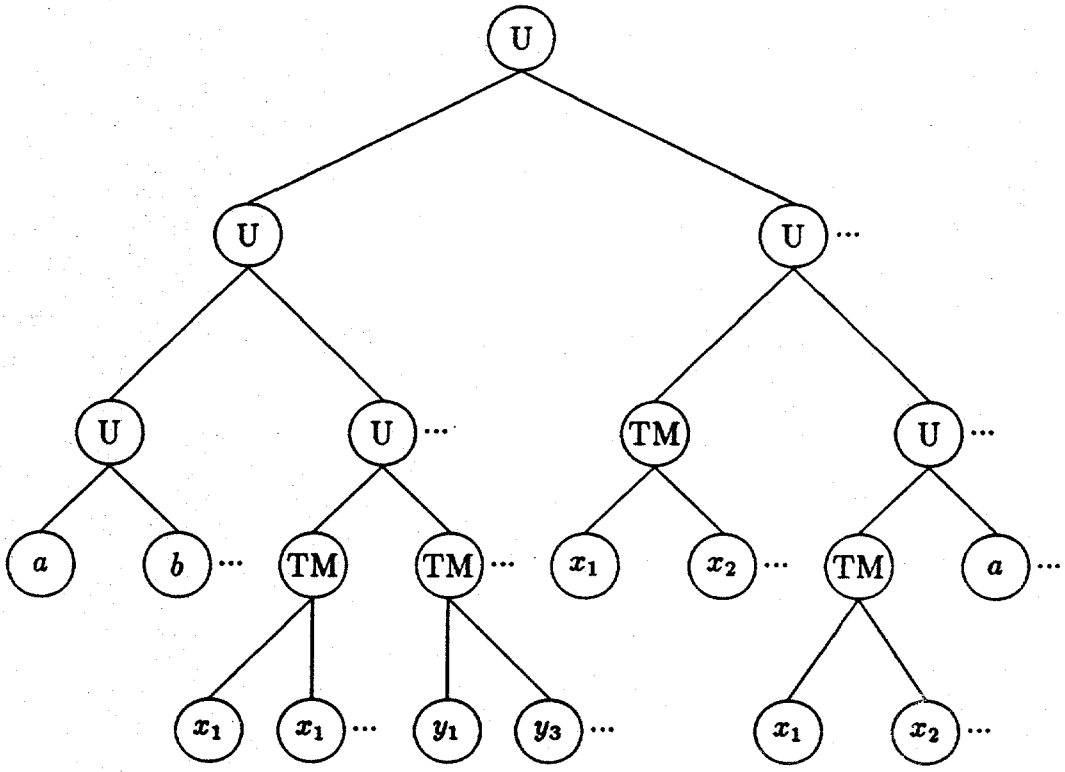


Figure 26.4: System B

non-replicated system A in some sense. That is,

$$A(\alpha) \overset{simulates}{\longleftarrow} B(\beta)$$

where the notion of simulation is that any schedule of B (denoted by β) looks the same to all user transactions as a schedule (α) of A. This can be stated formally as:

Theorem 26.1 $\forall\beta, \exists\alpha, \forall U, \alpha \mid U = \beta \mid U$

The order of the quantifiers makes this a very strong statement. It says that there exists a corresponding schedule of system A ($\exists\alpha$) for all users ($\forall U$). That is all user transactions think they are running in the *same* schedule of the serial system.

To complete our proof we also need a notion of transitivity for the simulations. That is,

$$B \overset{simulates}{\longleftarrow} C \Longrightarrow A \overset{simulates}{\longleftarrow} C.$$

This in turn can be stated formally as:

Theorem 26.2 $\forall\alpha\forall T, \exists\beta, \beta \mid T = \gamma \mid T \Longrightarrow \forall\gamma, \forall U, \exists\alpha, \alpha \mid U = \gamma \mid U$, *where* T *and* U *are non-orphan transactions in* γ.

This theorem follows immediately from Theorem 1. In this theorem γ refers to a behaviour of C, and T denotes a transaction. Note the weaker order of quantification.

The two theorems give us a very nice separation of concerns. We can use our usual serializability theorems to show the hypothesis of Theorem 2.

A proof of Theorem 26.1 needs a few preliminary definitions.

Define an *Access Sequence of* X *in* β ($Acess(X, \beta)$) as the set of CREATE and REQ-COMMIT actions of TM's for X in β. Define a $LogicalState(X, \beta)$ as the value of write-TM with last REQ-COMMIT action in access (X, β) We want read-TM's to always return the logical state. This is stated formally in the following lemma.

Lemma 26.3 *Let* β *be a schedule of B. If* β *ends in REQUEST-COMMIT* (T, v) *with* T *a read-TM for* X, *then* $v = LogicalState(X, \beta)$.

Proof: An easy induction, because B is a serial system. ∎

Given this lemma, Theorem 1 follows almost immediately.

26.4.3 Reconfiguration

In order to accomodate reconfiguration, each replica must now store the 4-tuple:

$$(value, VersionNumber, configuration, ConfigurationNumber).$$

The algorithm do a $READ_X$ is now as follows. First access (read) a read-quorum in the configuration with the highest configuration number seen; then return the value with the highest version number. Note that the highest configuration number seen may change as more replicas are read.

The algorithm to do a $WRITE_X$ is now as follows. First access (read) a read-quorum in the configuration with the highest configuration number seen exactly as for $READ_X$. Let $NewVn = HighestVersionNumberSeen + 1$; now do write accesses with $(v, NewVn)$ to a write quorum in the configuration with highest configuration number seen.

The algorithm to do a $RECONFIGURE$ is similar to a $WRITE$, except that in this case we must write to the configuration and configuration number components of an object, while incrementing highest-configuration-number-seen. This is actually an improvement over Gifford's scheme. In his original version, the new configuration was written to both an old and new write-quorum. This simplification was discovered when a certain precondition was not used to prove Gifford's algorithm correct – proofs are useful!

In order to prove the reconfiguration algorithm correct, it would be nice to introduce Reconfiguration-TMs as children of the user transactions so we can model the right atomicity. But that is difficult because we also want reconfiguration to be transparent to the user. A solution is to use Spy Automata which we compose with the user transactions. These spy on user activities and have the Reconfiuration-TMs nested below them. The final solution in [Goldman87] introduced another level of nesting beneath all the TMs for better modularity. This is illustrated in Figure 26.5 and 26.6. Please refer to [GoldmanL87] for more details.

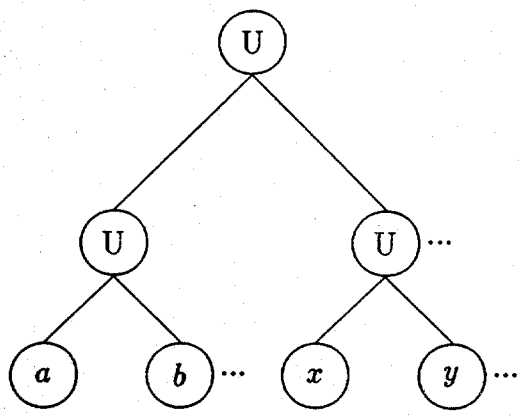


Figure 26.5: System A with Reconfiguration

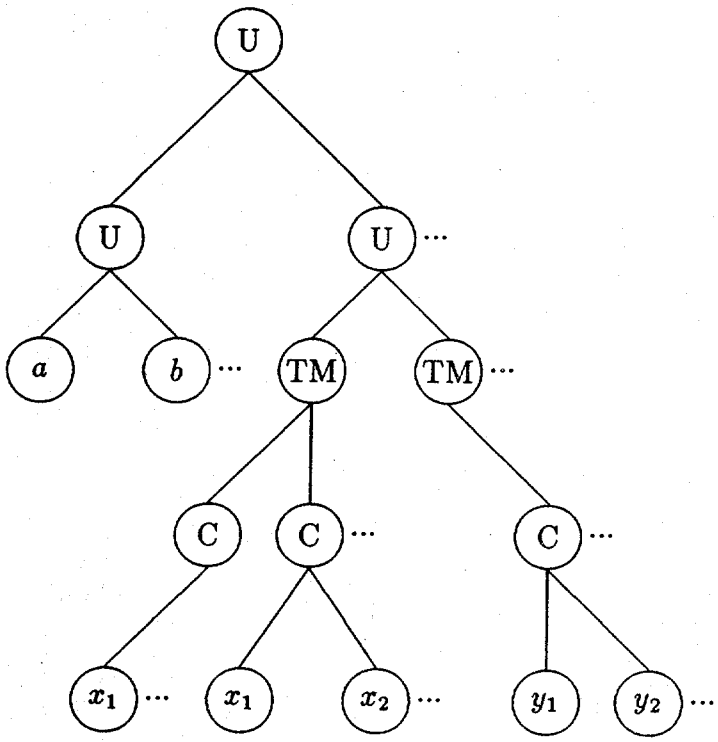


Figure 26.6: System B with Reconfiguration. A spy automaton, not shown, is composed with each user automaton. All reconfigure-TM's are invoked by the spy, and the results are reported back to the spy. Other TM's are invoked by the user automata as before.

Bibliography

[1] K. Abrahamson, A. Adler, L. Higham, and D. Kirkpatrick. *Probabilistic Solitude Detection I: Ring Size Known Approximately.* Technical Report 87-8, University of British Columbia, Vancouver, B.C., Canada, March 1987.

[2] K. Abrahamson, A. Adler, L. Higham, and D. Kirkpatrick. *Probabilistic Solitude Detection II: Ring Size Known Exactly.* Technical Report 87-11, University of British Columbia, Vancouver, B.C., Canada, April 1987.

[3] A. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proceedings of 28th IEEE Symposium on Foundations of Computer Science*, pages 358–370, October 1987.

[4] Y. Afek and E. Gafni. Time and message bounds for election in synchronous and asynchronous complete networks. In *Proceedings of 4th ACM Symposium on Principles of Distributed Computing*, pages 186–195, Minaki, Ontario, August 1985.

[5] A. Aho, J. Ullman, A. Wyner, and M. Yannakakis. Bounds on the size and transmission rate of communication protocols. *Computers and Mathematics with Applications*, 8(3):205–214, 1982.

[6] D. Angluin. Local and global properties in networks of processors. In *Proceedings of 12th ACM Symposium on Theory of Computing*, pages 82–93, 1980.

[7] E. Arjomandi, M. Fischer, and N. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the Association for Computing Machinery*, 30(3):449–456, July 1983.

[8] J. Aspnes, A. Fekete, N. Lynch, M. Merritt, and W. Weihl. A theory of timestamp-based concurrency control for nested transactions. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 431–444, August 1988.

[9] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. Manuscript.

[10] H. Attiya, M. Snir, and M. Warmuth. Computing in an anonymous ring. To appear in *Journal of the ACM.*

[11] B. Awerbuch. Complexity of network synchronization. *J. ACM*, 32(4):804–823, October 1985. Also, Technical Memo MIT/LCS/TM-268, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, January 1985.

[12] B. Awerbuch. Reducing complexities of distributed maximum flow and breadth-first search algorithms by means of network synchronization. *Networks*, 15:425–437, 1985.

[13] B. Awerbuch, O. Goldreich, D. Peleg, and R. Vainish. A tradeoff between information and communication in broadcast protocols. In *Proceedings of the Aegean Workshop on Computing*, 1988.

[14] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *Proceedings of the 29th IEEE Symposium on Foundations of Computer Science*, IEEE, October 1988.

[15] A. Bar-Noy, D. Dolev, C. Dwork, and H. Strong. Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 42–51, August 1987.

[16] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proceedings of 2nd ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[17] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1986.

[18] O. Biran, S. Moran, and S. Zaks. A combinatorial characterization of the distributed tasks which are solvable in the presence of one faulty processor. In *Proceedings of 7th ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[19] B. Bloom. Constructing two-writer atomic registers. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 249–259, Vancouver, British Columbia, Canada, August 1987. Also, to appear in special issue *IEEE Transactions On Computers.*

[20] G. Bracha. An $O(\log n)$ expected rounds randomized *Byzantine* generals algorithm. In *Proceedings of 17th Symposium on Theory of Computing*, page 316–326, May 1985. Journal of ACM, 34(4):910-92[illegible]87.

[21] G. Bracha and S. Toueg. A distributed algorithm for generalized deadlock detection. *Distributed Computing*, 2:127–138, 1987.

[22] M. Bridgeland and R. Watro. Fault tolerant decision making in totally asynchronous distributed systems. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987.

[23] J. Burns. *A formal model for message passing systems.* Technical Report TR-91, Computer Science Dept., Indiana University, May 1980.

[24] J. Burns, M. Fischer, P. Jackson, N. Lynch, and G. Peterson. Data requirements for implementation of n-process mutual exclusion using a single shared variable. *Journal of the ACM*, 29(1):183–205, 1982.

[25] J. Burns and N. Lynch. *The Byzantine Firing Squad Problem.* Technical Memo MIT/LCS/TM-275, Laboratory for Computer Science,Massachusetts Institute Technology, April 1985.

[26] J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of 18th Annual Allerton Conference on Communications, Control, and Computing*, pages 833–842, 1980.

[27] O. Carvalho and G. Roucairol. Assertion, decomposition and partial correctness of distributed control algorithms. *Distributed Computing Systems*, 67–93, 1983.

[28] O. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the ACM*, 26(2):146–148, 1983.

[29] K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[30] K. Chandy and J. Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, October 1984.

[31] K. Chandy and J. Misra. How processes learn. *Distributed Computing*, 1(1):40–52, 1986.

[32] K. Chandy, J. Misra, and L. Haas. Distributed deadlock detection. *ACM Transactions on Programming Languages and Systems*, 1(2):144–156, May 1983.

[33] K. M. Chandy and J. Misra. On proofs of distributed algorithms, with application to the problem of termination detection. Manuscript.

[34] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, 1988.

[35] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22:281–283, May 1979.

[36] B. Chor and B. Coan. A simple and efficient randomized Byzantine agreement algorithm. In *IEEE Transactions on Software Engineering*, pages 531–539, 1985. Also, revised in B. Coan ,*Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[37] B. Coan, D. Dolev, C. Dwork, and L. Stockmeyer. The distributed firing squad problem. In *Proceedings of the 17th ACM Symposium on Theory of Computing*, pages 335–345, May 1985.

[38] B. Coan and J. Lundelius. Transaction commit in a realistic fault model. In *Proceedings of 5th Annual ACM Symposium on Principles of Distributed Computing*, pages 40–51, Calgary, Alberta, Canada, August 1986.

[39] B.A. Coan. A communication-efficient canonical from for fault-tolerant distributed protocols. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 63–72, August 1986. Also, revised in B. Coan, *Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

[40] J.G. DeBruijn. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 10(3):137–138, 1967.

[41] E. Dijkstra and C. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1), August 1980.

[42] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 115–138, 1971.

[43] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8(9):569, September 1965.

[44] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3:14–30, 1982.

[45] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[46] D. Dolev, E. Gafni, and N. Shavit. Toward a non-atomic era: *l*-exclusion as a test case. In *Proceedings of 20th ACM Symposium on Theory of Computing*, pages 78–92, May 1988.

[47] D. Dolev, M. Klawe, and M. Rodeh. *An* $O(n \log n)$ *unidirectional distributed algorithm for extrema finding in a circle.* Research Report RJ3185, IBM, July 1981. *J. Algorithms*, 3:245–260, 1982.

[48] D. Dolev, N. Lynch, S. Pinter, E. Stark, and W. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):449–516, 1986.

[49] D. Dolev and H. Strong. Authenticated algorithms for Byzantine agreement. *SIAM J. Computing*, 12(4):656–666, November 1983.

[50] S. Dolev, J. Halpern, and R. Strong. On the possiblity and impossibility of achieving clock synchronization. In *Proceedings of 16th Symposium on Theory of Computing*, pages 504–510, May 1984. Journal of Computer and System Sciences, 32:230–250, 1986.

[51] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[52] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 1988. To appear.

[53] C. Dwork and Y. Moses. Knowledge and common knowledge in a *Byzantine environment I*: crash failures. In *Proceedings of Conference on Theoretical Aspects of Reasoning about Knowledge*, 1986. Also, to appear in *Information and Computation.*

[54] C. Dwork and D. Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pages 1–11, August 1983.

[55] M. Eisenberg and M. McGuire. Further comments on *Dijkstra's* concurrent programming control. *Communications of the ACM*, 15(11):999, 1972.

[56] A. ElAbbadi and S. Toueg. Maintaining availability in partitioned replicated databases. In *Proceedings of 5th ACM Symposium on Principles of Database Systems*, pages 240–251, 1986.

[57] A. Fekete, N. Lynch, M. Merritt, and W. Weihl. Commutativity-based locking for nested transactions. Technical Memo, MIT/LCS/TM-370, Massachusetts Institute Technology, Laboratory for Computer Science, August 1988.

[70] J. Gray. *Notes on Data Base Operating Systems.* Technical Report IBM Report RJ2183(30001), IBM, February 1978. (Also in Operating Systems: An Advanced Course, Springer-Verlag Lecture Notes in Computer Science #60.).

[71] J. Halpern, B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Proceedings of the 3rd ACM Symposium on Principles of Distributed Computing*, pages 89–102, August 1984.

[72] J.Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 50–61, 1984. Revised as IBM Research Report, IBM-RJ-4421.

[73] M. Herlihy, N. Lynch, M. Merritt, and W. Weihl. On the correctness of orphan elimination algorithms. In *17th IEEE Symposium on Fault-Tolerant Computing*, pages 8–13, 1987. Also, MIT/LCS/TM-329, MIT Laboratory for Computer Science, Cambridge, MA, May 1987. Revised version to appear in *Journal of the ACM.*

[74] M. P. Herlihy. Impossibility and universality results for wait-free synchronization. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, pages 276–290, August 1988.

[75] D. Hirschberg and J. Sinclair. Decentralized extrema-finding in circular configuarations of processes. *Communications of the ACM*, 23:627–628, November 1980.

[76] G. Ho and C. Ramamoorthy. Protocols for deadlock detection in distributed database systems. *IEEE Transactions on Software Engineering*, SE-8(6):554–557, November 1982.

[77] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[78] P. Humblet. A distributed algorithm for minimum weight directed spanning trees. *IEEE Transactions on Computers*, COM-31(6):756–762, 1983. MIT-LIDS-P-1149.

[79] D.E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[80] E. Korach, S. Moran, and S. Zaks. Tight lower and upper bounds for some distributed algorithms for a complete network of processors. In *In Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 199–207, 1984.

[81] L. Lamport. The mutual exclusion problem. *Journal of the ACM*, 33(2):313–326, 327–348, 1986.

[82] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

[83] L. Lamport. On interprocess communication. *Distributed Computing*, 1(1):77–85, 86–101, 1986. *Digital Systems Research* TM-8.

[84] L. Lamport. Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems*, 5(2):190–222, April 1983.

[85] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, 1978.

[86] L. Lamport. The weak Byzantine generals problem. *Journal of the ACM*, 30(3):669–676, 1983.

[87] L. Lamport and N. Lynch. Chapter on distributed computing. To appear in *Handbook of Theoretical Computer Science.*

[88] L. Lamport and P. Melliar-Smith. Byzantine clock synchronization. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 68–74, August 1984.

[89] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[90] G. LeLann. Distributed systems, towards a formal approach. In *IFIP Congress*, pages 155–160, Toronto, 1977.

[91] M. Loui and H. Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4:163–183, 1987.

[92] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77:1–36, 1988.

[93] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62(2-3):190–204, August/September 1984.

[94] N. Lynch. *I/O Automata: A Model for Discrete Event Systems.* Technical Memo MIT/LCS/TM-351, Massachusetts Institute Technology, Laboratory for Computer Science, March 1988. Also, in *22nd Annual Conference on Information Science and Systems*, Princeton University, Princeton, N.J., March 1988.

[95] N. Lynch. Upper bounds for static resource allocation in a distributed system. *Journal Of Computer And Systems Sciences*, 23(2):254–278, October 1981.

[96] N. Lynch, B. Blaustein, and M. Siegel. Correctness conditions for highly available replicated databases. In *Proceedings of 5th ACM Symposium on Principles of Distributed Computing*, pages 11–28, Calgary, Alberta, Canada, August 1986.

[97] N. Lynch and M. Fischer. On describing the behavior and implementation of distributed systems. *Theoretical Computer Science*, 13:17–43, 1981.

[98] N. Lynch, Y. Mansour, and A. Fekete. The data link layer: two impossibility results. In *Proceedings of 7th ACM Symposium on Principles of Distributed Computation*, pages 149–170, Toronto, Canada, August 1988. Also, Technical Memo MIT/LCS/TM-355, May 1988.

[99] N. Lynch and E. Stark. *A Proof of the Kahn Principle for Input/Output Automata.* Technical Memo MIT/LCS/TM-349, Massachusetts Institute Technology, January 1988.

[100] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of 6th ACM Symposium on Principles of Distributed Computation*, pages 137–151, August 1987. Expanded version available as Technical Report MIT/LCS/TR-387, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA., April 1987.

[101] S. Mahaney and F. Schneider. Inexact agreement: accuracy, precision, and graceful degradation. In *Proceedings of 4th ACM Symposium on Principles of Distributed Computing*, pages 237–249, August 1985.

[102] D. Menasce and R. Muntz. Locking and deadlock detection in distributed databases. *IEEE Transactions on Software Engineering*, SE-5(3):195–202, May 1979.

[103] D. Mitchell and M. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of 3rd ACM Symposium on Principles of Distributed Computing*, pages 282–284, Vancouver, B.C., Canada, August 1984.

[104] S. Moran and Y Wolfstahl. Extended impossibility results for asynchronous complete networks. *Information Processing Letters*, 26:145–151, 1987.

[105] Y. Moses and M. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:249–259, 1988.

[106] Y. Moses and O. Waarts. Coordinated traversal: $(t+1)$-round byzantine agreement in polynomial time. In *Proceedings of 29th Symposium on Foundations of Computer Science*, pages 246–255, October 1988.

[107] R. Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[108] S. Owicki and D. Gries. An axiomatic proof technique for parrallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[109] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[110] G. Peterson and M. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of 9th ACM Symposium on Theory of Computing*, pages 91–97, May 1977.

[111] G.L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, 1983.

[112] G.L. Peterson. An $O(n \log n)$ unidirectional distributed algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems*, 4:758–762, October 1982.

[113] M. Rabin and D. Lehmann. On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem. In *Proceedings of 8th ACM Symposium on Principles of Programming Languages*, pages 133–138, 1981.

[114] M.O. Rabin. N-process mutual exclusion with bounded waiting by 4 log N- shared variable. *Journal of Computation and Systems Sciences*, 25:66–75, 1982.

[115] M.O. Rabin. Randomized *Byzantine* generals. In *Proceedings of 24th Symposium on Foundations of Computer Science*, pages 403–409, November 1983.

[116] M. Raynal. *Algorithms for Mutual Exclusion.* M.I.T. Press, 1986.

[117] G. Ricart and A. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981. Corrigendum in *Communications of the ACM*, 24(9).

[118] S. Sarin and N. Lynch. Discarding obsolete information in a replicated database system. *IEEE Transactions on Software Engineering SE*, 13(1):39–47, January 1987.

[119] R. Schaffer. On the correctness of atomic multi-writer registers. Bachelor's Thesis, June 1988, Massachusetts Institute Technology. Also, Technical Memo MIT/LCS/TM-364, Lab for Computer Science, MIT, June 1988 and Revised Technical Memo MIT/LCS/TM-364, January 1989.

[120] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(1):23–35, 1983.

[121] T. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.

[122] M. Staskauskas. The formal specification and design of a distributed electronic funds-transfer system. *IEEE Transactions on Computers*, 37(12):515–528, December 1988.

[123] R. Turpin and B. Coan. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Information Processing Letters*, 18(2):73–76, 1984. Also, Technical Report MIT/LCS/TR-315, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA, April 1984. Also, revised in B. Coan, *Achieving consensus in fault-tolerant distributed computing systems*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology,1987.

[124] P. Vitanyi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *Proceedings 27th Annual IEEE Symposium on Theory of Computing*, pages 233–243, Toronto, Ontario, Canada, May 1986. Also, MIT/LCS/TM-314, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA., 1986. Corrigenda in *Proceedings of 28th Annual IEEE Symposium on Theory of Computing*, page 487, 1987.

[125] J.L. Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.

Index