LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/RSS 16
Research Seminar Series

DISTRIBUTED ALGORITHMS

Lecture Notes for 6.852

Fall 1990

Nancy A. Lynch

Isaac Saias

February 1992

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Distributed Algorithms

Lecture Notes for 6.852

Fall Semester, 1990

Nancy A. Lynch
Isaac Saias

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 23. 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 1990. 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.

Acknowledgments

This compilation of notes would not have been possible without the hard work of the following students: Robert D. Blumofe, Anna Charny, Ken Chiang, Andrew Chou, Ken Decanio, Rainer Gawlick, David Goldstone, Shen-wei Huang, Marc Leblanc, S. J. Lee, Philippe Park, Boaz Patt, Roberto Segala, Jory Tsai, Suwei Wu, David Wald and Mary Ellen Zurko.

Very special thanks also are due to all the people that went in terra incognita in 1988 and produced the first set of lecture notes. We are in particular specially indebted to Ken Goldman who was the the TA in 1988. We thank also all the students of 1988 who contributed indirectly (and in some instances directly) to the compilation of these notes: 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 Leslie Lamport and Baruch Awerbuch for their "guest" lectures.

Finally, we thank Anna Wiseman for her unfaltering energy and help throughout the class and Joanne Talbot for her help in the final editing.

Course Syllabus

1 Introduction

[1] L. Lamport and N. Lynch. Distributed Computing. Chapter of Handbook on Theoretical Computer Science. North-Holland. Also, appeared as Technical Memo MIT/LCS/TM-384, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1989.

2 Asynchronous Systems

2.2 Models and Proof Techniques

[1] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.

[2] 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.

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

[4] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

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

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

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

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

[9] 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.

2.2 Shared Memory Algorithms

2.2.1 Mutual Exclusion

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

[2] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.

[3] Kenneth Goldman and Nancy Lynch. Modelling Shared State in a Shared Action Model. *Proceedings 5th Annual IEEE Symposium on Logic in Computer Science*, pages 450–463, June 1990.

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

[5] J.G. 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] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion. Manuscript 1990.

[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 system. 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 Symosium 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.* Tech Memo MIT/LCS/TM-290, Laboratory for Computer Science, Massachusetts Institute Technology, Cambridge, MA 02139, October, 1985. Also, ACM Transactions on Programming Languages and Systems, Vol 11, No. 1, January 1989, pages 90-114.

[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 Computation and Systems Sciences*, 25:66–75, 1982.

2.2.2 Dining Philosophers

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

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

[3] 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.

2.2.3 Atomic Registers

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

[2] L. Lamport. On interprocess communication. *Distributed Computing*, 1(1):77–85, 86–101, 1986. *Digital Systems Research* TM-8.

[3] 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.*

[4] 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.

[5] Russel Schaffer and Bard Bloom. *On the Correctness of Atomic Multi-writer Registers.* Technical Memo MIT/LCS/TM–364, MIT Laboratory for Computer Science, June 1988.

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

[7] 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.

[8] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec, Canada, August 1990. Also, Technical Memo MIT/LCS/TM-429, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1990. Submitted to *Journal of the ACM.*

[9] Soma Chaudhuri and Jennifer Welch. *Bounds on the Costs of Register Implementations.* Technical Report TR90-025, University of North Carolina at Chapel Hill, June 1990.

[10] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast? To appear in 1990 Symposium on the Foundation of Computer Science.

[11] Karl Abrahamson. On Achieving Consensus Using a Shared Memory. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 291-302, Toronto, Canada, August 1988.

2.3 Aynchronous Message Passing Algorithms

2.3.1 Computing in Static Graphs

Computing in a Ring

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

[2] 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.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

Computing in Complete Graphs

[1] 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.

[2] 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.

Computing in Arbitrary Graphs

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

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

[3] 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.

2.3.2 Logical Time

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

2.3.3 Resource Allocation

[1] 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).

[2] O. Carvalho and G. Roucairol. Assertion, decomposition and partial correctness of distributed control algorithms. *Distributed Computing Systems*, 67–93, 1983.

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

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

2.3.4 Communication

Datalink Protocols

[1] 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.

[2] Joseph Y. Halpern and Lenore D. Zuck. A little knowledge goes a long way: simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 269–280, August 1987.

[3] 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.

[4] A. Fekete, N. Lynch, Y. Mansour, and J Spinelli. *The Data Link Layer: The Impossibility of Implementation in Face of Crashes.* Technical Memo MIT/LCS/TM-355.b, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1989. Submitted for publication.

[5] Da-Wei Wang and Lenore Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 73–83, August 1989.

[6] Ewan Tempero and Richard Ladner. Tight bounds for weakly bounded protocols. In *Proceedings of the 9^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 205–212, Quebec, Canada, August 1990.

[7] A. Fekete and N. Lynch. The need for headers: an impossibility result for communication over unreliable channels. Also, Technical Memo MIT/LCS/TM-428, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May, 1990. Submitted for publication. To appear in CONCUR. 1990.

[8] Yehuda Afek, Hagit Attiya, Alan Fekete, Nancy Lynch, Yishay Mansour, Da-Wei Wang, and Lenore Zuck. Reliable Communication over Unreliable Channels. Manuscript.

End-to-End Protocols

[1] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks (extended abstract). 1990.

[2] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization: an effecient technique for end-to end communication. 1990.

Broadcast

[1] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols. 1990. To appear in JACM.

2.3.5 Detection of Stable Properties

Termination

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

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

Global Snapshots

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

[2] M. Fischer, N. Griffeth, and N. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982. Also, in *Proceedings of IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1981, 33-38.

Deadlock

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

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

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

[4] 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.

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

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

[7] 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.

2.3.6 Consensus

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

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

[3] 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.

[4] 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.

[5] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3), July 1990.

[6] 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.

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

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

2.3.7 Fault-Tolerance

[1] Gil Neiger and Sam Toueg. *Automatically Increasing the Fault-tolerance for Distributed Algorithms.* Technical Report TR90-1081, Cornell University, January 1990.

2.3.8 Self-Stabilization

[1] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, November 1974.

[2] James Burns and Jan Pachl. Uniform self-stabilizing rings. *Journal of ACM*, 11(2):330–344, April 1989.

[3] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–101, Quebec, Canada, August 1990.

[4] James Burns, Mohamed Gouda, and Raymond Miller. *Stabilization and Pseudo-stabilization.* Technical report TR-90-13, University of Texas at Austin, May 1990.

[5] Mohamed Gouda and Nicholas Multari. *Stabilizing Communication Protocols.* Technical report TR-90-20, University of Texas at Austin, June 1990.

3 Synchronous Systems

3.1 Synchronous Message-Passing Algorithms

3.1.1 Computing in a Ring

[1] G. 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.

[2] H. Attiya, M. Snir, and M. Warmuth. Computing in an anonymous ring. *Journal of the ACM*, 35(4):845–876, October 1988.

3.1.2 Distributed Consensus

Basic Results

[1] 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.).

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

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

[4] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and Raymond Strong. Shifting gears: changing algorithms on the fly to expedite Byzantine agreement. June 27 1990.

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

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

[7] 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.

Number of Processes

[1] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3:14–30, 1982.

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

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

[4] D. 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.

Time

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

[2] 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.*

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

Communication

[1] 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.

[2] 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.

[3] Piotr Berman and Juan Garay. Cloture voting: n/4-resilient distributed consensus in t+1 rounds. 1990.

Randomized Algorithms

[1] 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.

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

[3] I. Saias and N. Lynch. An analysis of Rabin's randomized mutual exclusion algorithm. MIT/LCS/TM-462, December 1991.

[4] G. Bracha. An $O(\log n)$ expected rounds randomized *Byzantine* generals algorithm. In *Proceedings of 17th Symposium on Theory of Computing*, pages 316–326, May 1985. Journal of ACM, 34(4):910-920,1987.

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

[6] Benny Chor and Cynthia Dwork. Randomization in Byzantine agreement. *Advances in Computing Research*, 5:443–497, 1989.

Approximate Agreement

[1] 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.

Firing Squad

[1] 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.

[2] 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.

Commit

[1] 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.

[2] 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. [Nancy. Synchronous]

The Knowledge Approach

[1] 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.

[2] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. *Information and Computation*, 1988. To appear.

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

3.2 Synchronous vs. Asynchronous Systems

[1] E. Arjomandi, M. Fischer, and N. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[2] Baruch Awerbuch. Complexity of network synchronization. *Journal of ACM*, 32(4):804–823, October 1985.

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

[4] 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.

4 Timing-Based Systems

4.1 Models and Proof Techniques

[1] Michael Merritt, Francesmary Modugno, and Mark Tuttle. Time constrained automata. July 23 1990.

[2] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In *Proceedings of the* 9th *Annual ACM Symposium on Principles of Distributed Computing*, Quebec, Canada, August 1990. Expanded version: Technical Memo MIT/LCS/TM-412.b, Laboratory for Computer Science, Massachusetts Institute of Technology, December 1989. Submitted for publication.

4.2 Algorithms and Bounds

4.2.1 Synchronization

[1] 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.

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

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

[4] 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.

[5] 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.

[6] Hagit Attiya and Marios Mavronicolas Efficiency of Asynchronous vs. Semi-Synchronous Network submitted to *the 28th annual Allerton Conference on Communication, Control and Computing*, 1990

4.2.2 Resource Allocation

[1] H. Attiya and N. Lynch Time Bounds for Real-Time Process Control in the Presence of Timing Uncertainty In *Proceedings of the 10th IEEE Real-Time Systems Symposium*, pages 268-284, December 1989.

4.2.3 Communication

[1] Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding-faults. 1989.

4.2.4 Consensus

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

[2] Leslie Lamport. *The Pårt-Time Parliament.* Technical Memo 49, Digital Systems Research Center, September 1 1989.

[3] H. Attiya, C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty.

[4] Ray Strong, Danny Dolev, and Flaviu Cristian. New latency bounds for atomic broadcast. April 1990.

5 Concurrency Control

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

[2] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. Atomic transactions. Book in progress.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 1: September 11

Lecturer: Nancy A. Lynch *Scribe: David Wald*

1.1 Introduction to the Course

1.1.1 What are Distributed Algorithms?

The heading of *Distributed Algorithms* includes algorithms for many different models of concurrency. The possible characteristics of these models include

- Shared Memory;
- Message Passing;
- Synchronous, Asynchronous and Partially Synchronous processing;
- Dataflow; and
- Databases

among others. One type of algorithm that is specifically excluded from this course is the sort of tightly-coupled parallelism in which all the processes are programmed together to solve a single problem, with the processes generally working over a tight, reliable network; this sort of model is covered in depth in Charles Leiserson and Tom Leighton's courses 6.848/18.435J and 6.849/18.436J. In contrast, the loosely-coupled parallelism covered in this class is characterized by *independence of activity*, including

- independent inputs and outputs at multiple locations;
- several processes executing at once under independent control;
- independent creation and termination of processes;
- occurrence of failures; and
- timing assumptions.

Above all, the hallmark of these algorithms is dealing with *uncertainty*.

These algorithms are often very complex. Even when the actual program code is short, the complications of distributed execution can make analysis extremely difficult. Rather than attempting to describe everything about the behavior of an algorithm, one basic method we will use is to attempt to prove certain global assertions about the algorithms execution.

Among the complications an algorithm might have to deal with are

- synchronization;
- the many possible interleavings of internal actions;
- inherent nondeterminism;
- action at multiple sites;
- order of interactions with the surrounding environment;
- failures;
- timing, including constraints on the duration of events; and
- continued operation (*e.g.*, operating systems).

We will need formal models if we want to precisely describe problems or algorithms to solve problems, and even more if we want to prove the correctness, complexity or impossibility of solutions to problems. Clearly any model we use must be able to account for the above complications. Unfortunately, there's no single accepted model in the literature of distributed algorithms, which may be why there's no textbook for the field. The model we'll be using in the course is that of I/O Automata.

1.1.2 Structure of the Course

The following outline closely parallels the outline of the course bibliography (Handout 2). In general, the course material is divided according to timing assumptions. The three basic categories covered are:

- *Asynchronous*: separate components take steps in arbitrary order.
- *Synchronous*: components take steps simultaneously.
- *Partially Synchronous (timing-based)*: in between, with some restrictions.

Within each category we'll consider several different problem domains.

2. Asynchronous Systems

2.1. Models and Proof Techniques

We'll begin with two lectures introducing I/O automata as a model for asynchronous systems, and showing how the model can be used to describe systems, specify properties of systems, and construct correctness proofs. The techniques used for the proofs include *invariant assertions*, *mappings*, and *liveness* and *safety* properties. We'll discuss a couple of typical examples (leader election and the Alternating Bit Protocol).

We'll then move on to asynchronous algorithms, dividing these into Shared Memory and Message Passing algorithms. Both of these styles can be modeled using I/O automata.

2.2. Shared Memory

The earliest work in this field arose from work on multiprocessing operating systems for uniprocessors. While these had only simulated parallelism, the concept of a process was found to be useful in programming. Research continues on shared memory algorithms, thanks to newer machines which, while truly parallel, continue to use a shared memory space available to all processors.

2.2.1. Mutual Exclusion

This very basic problem concerns the control of access to a single resource. We'll discuss the major algorithms, starting with the basic algorithm of Dijkstra. This simple problem will introduce many of the ideas mentioned above, including *progress*, *fairness*, *fault-tolerance*, *randomization*, and analysis of upper and lower bounds on time and memory.

2.2.2. Dining Philosophers

We'll move on to this more general resource allocation problem.

2.2.3. Atomic Registers

This topic deals with implementing shared memory while making weaker assumptions about the actual primitives we have available. One characteristic of these implementations is that they can be *wait-free*, never requiring one process to wait for another in order to synchronize.

2.3. Message Passing

Message Passing algorithms operate over a network. They can be modeled as graphs with processes at the nodes, and communication taking place by means of messages sent over the edges. This model is more loosely-coupled than the shared memory model, and thus allows both more independence and more uncertainty.

2.3.1. Computing in Static Graphs

Assume a fixed graph with fixed inputs and no failures, and compute something over this network.

2.3.2. Timestamps

2.3.3. Resource Allocation
This includes the Mutual Exclusion and Dining Philosophers problems.

2.3.4. Communication
Baruch Awerbuch's course 6.855/18.438J covers this in great detail, so we'll just touch on a few basic and interesting results, including *datalink*, *end-to-end* and *broadcast* protocols.

2.3.5. Detection of Stable Properties
Designing one algorithm to monitor certain properties of another, including checking for *termination* or *deadlock*, or taking a *global snapshot* of the current machine state.

2.3.6. Consensus
Getting processes to agree. This is easy in the absence of failures, but when we add failures we get some of our first impossibility proofs.

2.3.7. Self-Stabilization
Making a protocol that can recover from an arbitrary state.

3. Synchronous Systems

This is a simpler model, in which actions take place in a neat, round-by-round fashion. It might be a higher-level view, describing a system implemented on top of an asynchronous system. Note that this is not the same as a Parallel Random Access Machine (PRAM) model, since our synchronous systems usually communicate by message passing rather than shared memory, and can include the possibility of failures.

3.1. Typical Problems

3.1.1. Computing Functions on a Ring
Leader election and other problems.

3.1.2. Consensus
This is much easier than on asynchronous systems, and is often solvable (though not necessarily easy) in the presence of failures. The failures that we can deal with include processes stopping, and Byzantine failure, in which processes can do arbitrary, possibly malicious, communication. (The latter problem is closely related to material covered in Silvio Micali's courses 6.875/18.425J and 6.876/18.426J.)
We'll look at several results in this area, involving different kinds of resource restrictions.

3.2. Synchronous vs. Asynchronous Systems
How are the two models related, in terms of the problems that can be solved on each? In particular, we'll compare a result by Awerbuch, showing that a synchronous systems can be simulated efficiently by asynchronous systems, with

an apparently conflicting result by Arjomandi, Fischer and Lynch and we'll discuss the differences in the models involved.

4. Timing-Based Systems

   These systems fall between synchronous and asynchronous systems, allowing bounds to be specified for the time taken by process steps, message delivery, and possibly clocks, if we incorporate these into our models. We'll again cover problems including *synchronization*, *resource allocation*, *communication*, and *consensus.*

5. Concurrency Control

   This will receive little or no time in the course, in order to make time for timing-based systems. However, we may cover it to some extent, if there is sufficient interest.

1.1.3 Summary

In general, the course is intended to provide

- familiarity with many of the most important distributed algorithms and impossibility proofs;
- familiarity with some of the models and proof techniques;
- appreciation for the characteristic difficulties of designing such algorithms; and
- perhaps inspiration toward further work. There's a lot to be done.

So, let's begin...

1.2 Asynchronous Systems: Models and Proof Techniques

Distributed protocols can easily get complicated, even when there's very little code involved, and it's very easy (and fairly common) to make mistakes when reasoning about them. For this reason, it's useful to have a formal model to reason with. Many models have been proposed (about one for each researcher), several of which are listed in the bibliography. In this course we'll be using I/O Automata.

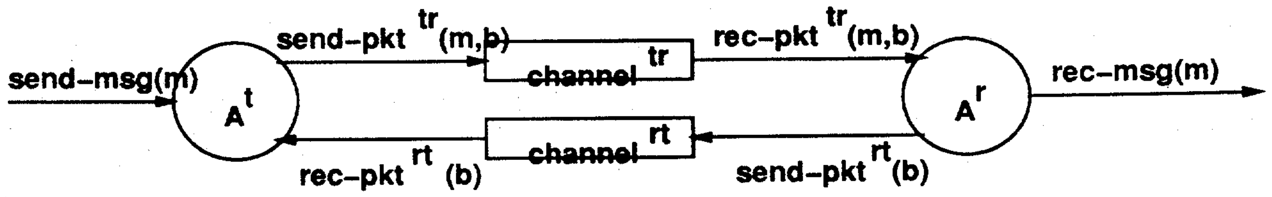


Figure 1.1: Network for the Alternating Bit Protocol

1.2.1 I/O Automata

I/O automata are similar to the automata in automata theory, being defined set-theoretically in terms of states, actions and transitions. The model has no concrete syntax, and thus differs from many other models, like Unity, CSP, and temporal logic, which are defined in terms of a basic syntax and syntactic transformation rules. This syntax-independence allows I/O automata to be used to describe algorithms in many different languages, as long as there is a way to extract the definitions of the automata from the program code. In general, this can make correctness arguments much simpler.

A full definition of I/O automata is given in Handout 4. We will go into more detail in the next lecture, but we'll first look at an example of a simple distributed algorithm, the Alternating Bit Protocol. (The following is essentially the same as Handout 5).

1.2.2 The Alternating Bit Protocol

Say we have two automata, A^t and A^r, and two channels, $channel^{tr}$ and $channel^{rt}$, between them (see Figure 1.1). Further assume that the channels are FIFO (first in, first out), but unreliable. That is, messages that are sent may or may not arrive, but any two messages that *do* arrive will arrive in the same order in which they were sent. The problem is to design an algorithm so that all messages sent by A^t will be reliably received by A^r.

The Alternating Bit Protocol is designed to give reliable data link level message delivery, for a given message alphabet M. The input actions are $send\_msg(m)$, $m \in M$, and the output actions are $receive\_msg(m)$, $m \in M$. That is, the interface of the protocol to the outside world is:

Inputs:
$send\_msg(m), m \in M$
Outputs:
$receive\_msg(m), m \in M$

Architecture

The architecture for an implementation consists of a *transmitter* automaton A^t, a *receiver* automaton A^r, and two unreliable FIFO physical channels, $channel^{tr}$ and $channel^{rt}$. The channel $channel^{tr}$ has input actions $send\_pkt^{tr}(m, b)$ and output actions $receive\_pkt^{tr}(m, b)$, where $m \in M$ and b is a Boolean. The channel $channel^{rt}$ has input actions $send\_pkt^{rt}(b)$ and output actions $receive\_pkt^{rt}(b)$, where b is a Boolean. The system is modeled by the composition of these automata, with all actions except $send\_msg(m)$ and $receive\_msg(m)$ hidden.

Channels

The channels are fairly ordinary FIFO queues, except that the effect of a $send\_pkt^{tr}$ or $send\_pkt^{rt}$ action is to put any finite number (possibly zero) copies of the data packet at the end of the queue. The effect of a $receive\_pkt^{tr}$ or $receive\_pkt^{rt}$, however, is always to remove exactly one copy. Moreover, if infinitely many "packets" are sent, then infinitely many are received. This can be described as the behavior of an I/O automaton (see Figures 1.2 and 1.3).

The Algorithm

The algorithm itself is fairly simple. In essence, A^t will send a packet, along with an identifying bit, and continue sending that packet with its bit until it receives that bit back from A^r, whereupon it will begin sending the next packet with a different bit. A^r, in turn, will remember the bit of the last packet in received, and continue sending that bit until it receives a packet with a different identifying bit, which it can consider to be a new packet. Thus, both A^t and A^r keep sending their last message until they receive confirmation that their message has been received.

The Description

The automata described in Figures 1.2 and 1.3 are presented in several parts, each describing a different aspect of an automaton. The *interface* lists the input and output actions of an automaton that are "visible" to the external system. The *state* contains the internal variables that, taken together, can be considered to be the state of the automaton. The *steps* section gives a more detailed description of the behavior of the actions, including any preconditions and internal effects of each of them. Any step is considered able to act at any time, provided its preconditions are satisfied. The *partitions* divide the output actions into sets, each of which must be given a fair chance to execute (for some definition of "fair," to be given later). This gives a way of insuring that no one action is executed constantly, locking out all others.

Interface:

Inputs:
$send\_msg(m), m \in M$
$receive\_pkt^{rt}(b), b$ a Boolean
Outputs:
$send\_pkt^{tr}(m, b), m \in M, b$ a Boolean

The state consists of the following components:
buffer, a finite queue of elements of M, initially empty, and
flag, a Boolean, initially 1.

The steps are:

```
$send\_msg(m), m \in M$
    Effect:
        add $m$ to *buffer*.

$send\_pkt^{tr}(m, b), m \in M, b$ a Boolean
    Precondition:
        $m$ is first on *buffer*.
        $b = flag$
    Effect:
        None.

$receive\_pkt^{rt}(b), b$ a Boolean
    Effect:
        if $b = flag$ then
            [remove first element (if any) from *buffer*;
            $flag := flag + 1 \mod 2$]
```

Partition:
all $send\_pkt^{tr}$ actions are in one class.

Figure 1.2: ABP Transmitter A^t

Interface:

Inputs:
  $receive\_pkt^{tr}(m, b), n \in M, b$ a Boolean
Outputs:
  $receive\_msg(m), m \in M$
  $send\_pkt^{rt}(b), b$ a Boolean

State:
  buffer, a finite queue of elements of M, initially empty, and
  flag, a Boolean, initially 0.

Steps:

```
$receive\_msg(m), m \in M$
    Precondition:
        $m$ is first on $buffer$.
    Effect:
        remove first element from $buffer$.

$receive\_pkt^{tr}(m, b), m \in M, b$ a Boolean
    Effect:
        if $b \neq flag$ then
            [add $m$ to $buffer$;
            $flag := flag + 1 \bmod 2$]

$send\_pkt^{rt}(b), b$ a Boolean
    Precondition:
        $b = flag$
    Effect:
        None.
```

Partition:
  all $send\_pkt^{rt}$ actions are in one class, and
  all $receive\_msg$ actions are in another class.

Figure 1.3: ABP Receiver A^r

This description of the automata corresponds to the technical definitions described below, and gives a fairly clean way of describing a protocol.

1.2.3 ABP with Sequence Numbers (ABP-S)

This is a simple variant of the ABP that uses sequence numbers instead of bits (see Figures 1.4 and 1.5). As before, the transmitter continues to send the same message just until it receives an acknowledgment with that message's tag; then it goes on to the next message. The receiver, on the other hand, keeps acknowledging the last message it has received, just until it gets the next message. It's presented here because it may be a little easier to prove correct than ABP, and it has a simple relationship to ABP.

The channels used here are the same as for the ABP, except that they transmit packets with integer tags rather than Boolean tags.

Interface:

Inputs:
  $send\_msg(m), m \in M$
  $receive\_pkt^{rt}(i), i$ a nonnegative integer
Outputs:
  $send\_pkt^{tr}(m, i), m \in M, i$ a nonnegative integer

State:
  buffer, a finite queue of elements of M, initially empty, and
  integer, a nonnegative integer, initially 1.

Steps:

```
$send\_msg(m), m \in M$
    Effect:
        add $m$ to $buffer$.

$send\_pkt^{tr}(m, i), m \in M, i$ a nonnegative integer
    Precondition:
        $m$ is first on $buffer$.
        $i = integer$
    Effect:
        None.

$receive\_pkt^{rt}(i), i$ a nonnegative integer
    Effect:
        if $i = integer$ then
            [remove first element (if any) from $buffer$;
            $integer := integer + 1$]
```

Partition:
  all $send\_pkt^{tr}$ actions are in one class.

Figure 1.4: ABP-S Transmitter A^t

Interface:

Inputs:
  $receive\_pkt^{tr}(m, i), n \in M, i$ a nonnegative integer
Outputs:
  $receive\_msg(m), m \in M$
  $send\_pkt^{rt}(i), i$ a nonnegative integer

State:
  buffer, a finite queue of elements of M, initially empty, and
  integer, a nonnegative integer, initially 0.

Steps:

```
receive_msg(m), m ∈ M
    Precondition:
        m is first on buffer.
    Effect:
        remove first element from buffer.

receive_pkt^tr(m, i), m ∈ M, i a nonnegative integer
    Effect:
        if i = integer + 1 then
            [add m to buffer;
            integer := integer + 1]

send_pkt^rt(i), i a nonnegative integer
    Precondition:
        i = integer
    Effect:
        None.
```

Partition:
  all $send\_pkt^{rt}$ actions
  all $receive\_msg$ actions

Figure 1.5: ABP-S Receiver A^r

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 2: September 13

Lecturer: Nancy A. Lynch *Scribe: Jory Tsai*[1]

2.1 I/O Automata

2.1.1 Introduction

The *Input/Output Automaton* (I/O Automata) model has been defined as a tool for modeling concurrent and distributed discrete event systems 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.

2.1.2 Overview of the Model

I/O automata provide an appropriate model for discrete event systems consisting of concurrently-operating components. Such systems continuously receive input from and react to their environment.

We list the goals, characteristics, and properties of the I/O Automata model before we start looking into more details.

- The primary goal behind the design of the I/O Automata model is to get a unified way of expressing distributed algorithms .
- I/O also allow to support system complexity analysis, to evaluate upper and lower bounds in the system's and components' efficiency, and to prove impossibility results. The model also can be used as a formal basis for algorithm correctness proofs–proofs that particular algorithms solve particular problems.
- **Characteristics and Properties:**
  1. *Inputs are always enabled* – This fundamental property of the model is about the way an automaton relates to its environment. For a given automaton A, an action is considered a *local* action if its being enabled depends solely of the state of A. Otherwise it is classified as an *input* action to the automaton A

[1]Based on lecture notes from 1988 scribed by John Keen.

otherwise. The assumption of the model is then that input actions are unable to be blocked by any automaton: the I/O Automata model does not allow an automaton to block its environment or eliminate undesirable inputs. Suppose we wish to guarantee that an automaton exhibits some behavior only when the environment observes certain restrictions on the production of inputs. Instead of allowing the automaton to block the bad inputs, we permit these inputs to occur, but also permit the automaton to exhibit arbitrary behavior when they do.

2. *Correctness* – model correctness conditions can often be expressed loosely as being of the form "if the environment behaves correctly, then the automaton behaves correctly". Alternatively, our correctness condition may require the automaton to delete bad inputs and response to them with error messages.
3. *Nondeterminism* – I/O automaton 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.
4. *Composition* – we can compose a collection of automata: first, we identify the same-named actions of the different automata. The composition guarantees that if one automaton has π as an output action, then π is an input action of all remaining automata having π as an action. As a result, an automaton generating an output action does so autonomously, and this output is transmitted instantaneously to all other automata having the same action as an input. All such components are passive recipients of the input, and react simultaneously with the output action.
5. *Fairness* – When I/O automata are run, they generate "executions" (alternating sequences of states and actions). 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 behavior corresponding to a fair execution of an automaton will be called also a fair behavior. It is defined as being the subsequence of the fair execution consisting of the external (input and output) actions.
6. *Problem Specification* -We can define a problem as a set of "allowable" behaviors. A machine will be said to "solve" a given problem if its behaviors are within that given set.
7. *Abstraction Mapping* – the model permits description of algorithms and systems at different levels of abstraction. Informally, abstraction mappings map automata that include implementation detail to more abstract automata that suppress some

of the details. 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 if A solves problem P, then B also solves P.

2.1.3 Formal description of the Input/Output Automaton Model

In this section we formally define the I/O Automata model of computation. We then show how it can be used to model a system, and to prove that a given I/O Automata system satisfies some specification.

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) ∪ out(S) and refer to the actions in ext(S) as the *external actions* of S. Also, we let *local(S)* = out(S) ∪ int(S), and refer to the actions in local(S) as the *locally-controlled actions* of S. Finally, we let *acts(S)* = *in(S)* ∪ *out(S)* ∪ *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.

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)*. 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)*.

2.1.4 Composition

As a motivation, let's consider the ABP model. We can think of this model as being the construction of four automata. We already saw two of them: the transmitter and the receiver. Both of them were described as I/O automata. The other two are two unreliable FIFO channels. These two could also be given as I/O automata. (But in describing them as such, we must be careful to make sure that their fair behaviors satisfy the liveness property: a channel has to deliver eventually something! The notions of behavior and of fairness will be introduced formally in the sequel.) Hence we see that the ABP model can be thought of as an I/O automaton, being itself the composition of four more primitive I/O automata.

Generally speaking, we can construct an automaton modeling a complex system by composing automata modeling the simpler system components. The essence of this composition is quite simple: when we compose a collection of automata, we identify an output action π of one automaton with the input action π of each automaton having π as an input action. Consequently, when one automaton having π as an output action performs π , all automata having π as an input action perform π simultaneously (automata not having π as an action do nothing).

We impose certain restrictions on the composition of automata. Since internal actions of an automaton A are intended to be unobservable by any other automaton B, we cannot allow A to be composed with B unless the internal actions of A are disjoint from the actions of B, since otherwise one of A's internal actions could force B to take a step. Furthermore, in keeping with our philosophy that at most one system component controls the performance of any given action, we cannot allow A and B to be composed unless the output actions of A and B form disjoint sets. Finally, since we do not preclude the possibility of composing a countable collection of automata, each action of a composition must be an action of only finitely many of the composition's components. Note that with infinite products we can handle systems that can create processes dynamically.

Since the action signature of a composition (the composition's interface with its environment) is determined uniquely by the action signatures of its components, it is convenient to define a composition of action signatures before defining the composition of automata. The preceding discussion motivates the following definition. A countable collection $S_{i_{i \in I}}$ of action signatures is said to be *strongly compatible* if for all $i, j \in I$ satisfying $i \neq j$ we have

1. $out(S_i) \cap out(S_j) = \emptyset$,
2. $int(S_i) \cap acts(S_j) = \emptyset$, and
3. no action is contained in infinitely many sets $acts(S_i)$.

We say that a collection of automata are *strongly compatible* if their action signatures are strongly compatible.

When we compose a collection of automata, internal actions of the components become internal actions of the composition, output actions become output actions, and all other actions (each of which can only an input action of a component) become input actions.

As motivation for this decision, consider one automaton A having π as an output action and two automata B_1 and B_2 having π as an input action. Notice that π is essentially a broadcast from A to B_1 and B_2 in the composition $A \cdot B_1 \cdot B_2$ of the three automata. Notice, however, that if we hide communication, then the composition $(A \cdot B_1) \cdot B_2$ would not be the same as the composition $A \cdot B_1 \cdot B_2$ since π would be made internal to $A \cdot B_1$ before composing with B_2, and hence π would no longer be a broadcast to both B_1 and B_2. This is problematic if we want to reason about the system $A \cdot B_1 \cdot B_2$ in a modular way by first reasoning about $A \cdot B_1$ and then reasoning about $A \cdot B_1 \cdot B_2$. We will define another operation to hide such communication actions explicitly.

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

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

As an illustration consider the ABP model being the composition of four different automata. The output actions of this composition are all the actions of the components except *send_msg* which are the only input actions; the internal actions are $send\_pkt^{rt}$, $send\_pkt^{tr}$, $receive\_pkt^{rt}$, $receive\_pkt^{tr}$. When for instance a $send\_pkt^{tr}$ occurs, it is atomically "shared" by the transmitter and the channel $channel^{tr}$.

The *composition* $A = \prod_{i \in I} A_i$ of a countable collection of strongly compatible automata $\{A_i\}_{i \in I}$ is the automaton defined as follows:[2]

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

When I is the finite set $1, ..., n$, we often denote $\prod_{i \in I} A_i$ by $A_1 \cdot \dots \cdot A_n$.

Notice that since the automata A_i are input-enabled, so is their composition. The partition of the composition's locally-controlled actions is formed by taking the union of the components' partitions (that is, each equivalence class of each component becomes an equivalence class of the composition).

Three basic results relate the executions, schedules, and behaviors of a composition to those of the composition's components. The first says, for example, that an execution of a composition induces executions of the component automata. Given an execution $\alpha = \vec{s_0}\pi_1\vec{s_1}\dots$ 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 $\vec{s_j}[i]$.

[2]Here *start*(A) and *states*(A) are defined in terms of the ordinary Cartesian product, while *sig*(A) is defined in terms of the composition of actions signatures just defined. Also, we use the notation $\vec{s}[i]$ to denote the ith component of the state vector $\vec{s}$.

Proposition 3 *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 every $i \in I$. Moreover, the same result holds if execs is replaced by it finexecs, scheds, finscheds, behs, or finbehs.*

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

Proposition 4 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $acts(A)$ such that $\beta|A_i = sched(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$. Moreover, the same result holds when acts and sched are replaced by ext and beh, respectively.*

As a corollary, schedules and behaviors of component automata can also be pasted together to form schedules and behaviors of the composition.

Proposition 5 *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 every $i \in I$, then $\beta \in scheds(A)$. Moreover, the same result holds when acts and scheds are replaced by ext and behs, respectively.*

As promised, we now define an operation that 'hides' actions of an automaton by converting them to internal actions. This operation is useful for redefining what the external actions of a composition are. We begin with a hiding operation for action signatures: if S is an action signature and $\Sigma \subseteq acts(S)$, then $hide_\Sigma S = S'$ where $in(S') = in(S) - \Sigma$, $out(S') = out(S) - \Sigma$ and $int(S') = int(S) \cup \Sigma$. We now define a hiding operation for automata: if A is an automaton and $\Sigma \subseteq acts(A)$, then $hide_\Sigma A$ is the automaton A' obtained from A by replacing $sig(A)$ with $sig(A') = hide_\Sigma sig(A)$.

5.0.5 Fairness

We are in general only interested in the executions of a composition in which all components are treated fairly. While what it means for a component to be treated fairly may vary from context to context, it seems that any reasonable definition should have the property that infinitely often the component is given the opportunity to perform one of its locally-controlled actions. In this section we define such a notion of fairness.

As we have mentioned, the partition of an automaton's locally-controlled actions is intended to capture some of the structure of the system the automaton is modeling. Each class of actions is intended to represent the set of locally-controlled actions of some system component.

The definition of automaton composition guarantees that an equivalence class of a component automaton becomes an equivalence class of a composition, and hence that composition retains the *essential* structure of the system's primitive components.[1] In our model, therefore, being fair to each component means being fair to each equivalence class of locally-controlled actions. This motivates the following definition.

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 α contains infinitely many occurrences of states in which no action of C is enabled.

This says that a fair execution gives fair turns to each class C of $part(A)$, and therefore to each component of the system being modeled. Infinitely often the automaton attempts to perform an action from the class C. On each attempt, either an action of C is performed, or no action from C *can* be performed since no action from C is enabled. For example, we may view a finite fair execution as an execution at the end of which the automaton repeatedly cycles through the classes in round-robin order attempting to perform an action from each class, but failing each time since no action is enabled from the final state.

We denote the set of fair executions of A by $fairexecs(A)$. 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 $fairscheds(A)$. 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 $fairbehs(A)$.

Let's give some examples. First let's consider executions of the automaton A^t of the ABP model.

The execution consisting of the single state (empty buffer, 1) is a fair (trivial!) execution: nothing is enabled. The two following behaviors are also fair behaviors of A^t:
$send\_msg(m)\ receive\_pkt^{rt}(0)\ send\_pkt^{tr}(m,1)\ send\_pkt^{tr}(m,1)\ send\_pkt^{tr}(m,1)\ldots$
$send\_msg(m) receive\_pkt^{rt}(0)\ send\_pkt^{tr}(m,1)\ send\_pkt^{tr}(m,1)\ receive\_pkt^{rt}(1)$.
but not:
$send\_msg(m)\ receive\_pkt^{rt}(0)\ send\_pkt^{tr}(m,1)\ send\_pkt^{tr}(m,1)\ send\_pkt^{tr}(m,1)$.

Let's turn to A^r. Then:
$receive\_pkt^{rt}(m,1)\ send\_pkt^{rt}(1)\ send\_pkt^{rt}(1)\ receive\_msg(m)\ send\_pkt^{rt}(1)\ldots$ is a fair behavior; but not

[1]It might be argued that retaining this partition is a bad thing to do since it destroys some aspects of abstraction. Notice, however, that any reasonable definition of fairness must lead to some breakdown of abstraction since being fair means being fair to the primitive components which must somehow be modeled.

$receive\_pkt^{rt}(m,1)$ $send\_pkt^{rt}(1)$ $send\_pkt^{rt}(1)$ $send\_pkt^{rt}(1)\ldots$ or
$receive\_pkt^{rt}(m,1)$ $send\_pkt^{rt}(1)$ $send\_pkt^{rt}(1)$ $send\_pkt^{rt}(1)$ $receive\_msg(m)$.

Let's return to our general exposition. We can prove the following analogues to Propositions 1-3 in the preceding section:

Proposition 6 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. If $\alpha \in fairexecs(A)$ then $\alpha|A_i \in fairexecs(A_i)$ for every $i \in I$. Moreover, the same result holds if fairexecs is replaced by fairscheds or fairbehs.*

Proposition 7 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. Suppose α_i is a fair execution of A_i for every $i \in I$, and suppose β is a sequence of actions in acts(A) such that $\beta|A_i = sched(\alpha_i)$ for every $i \in I$. Then there is a fair execution α of A such that $\beta = sched(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$. Moreover, the same result holds when acts and sched are replaced by ext and beh, respectively.*

Proposition 8 *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 $\beta|A_i \in fairscheds(A_i)$ for every $i \in I$, then $\beta \in fairscheds(A)$. Moreover, the same result holds when acts() and fairscheds() are replaced by ext and fairbehs, respectively.*

We state these results because analogous results often do not hold in other models. As we will see in the following section, the fact that the fair behavior of a composition is uniquely determined by the fair behavior of the components makes it possible to reason about the fair behavior of a system in a modular way.

8.0.6 Problem Specification

We want to say that a problem specification is simply a set of allowable 'behaviors,' and that an automaton solves the specification if each of its 'behaviors' is contained in this set. The automaton solves the problem in the sense that every 'behavior' it exhibits is a 'behavior' allowed by the problem specification (but notice that there is no single 'behavior' the automaton is *required* to exhibit). The appropriate notion of 'behavior' (e.g., finite behavior, infinite behavior, fair behavior, etc.) used in such a definition depends to some extent on the nature of the problem specification.

Safety properties are informally characterized by the fact that they specify a property that must hold in every state of a computation. Since an infinite computation satisfies a safety property if and only if every finite prefix of the computation does so, the notion of 'behavior' most useful in this context seems to be finite behaviors.

Liveness properties are informally characterized by the fact that they specify events that must eventually be performed. A reliable candy machine, for example, should satisfy the liveness condition that if a button is pushed, then a candy bar (of the correct type) is eventually dispensed. Clearly this is a property of infinite behaviors, and not finite behaviors. In fact, this is a property that can only be satisfied by fair behaviors, since a candy machine cannot dispense the required candy bar if it is not given the chance to do so. The notion of 'behavior' most useful in this context, therefore, seems to be fair behaviors.

Consequently, we would like to say that a specification is a set of allowable behaviors, and that an automaton solves the specification if all finite or fair behaviors (depending on the context) of the automaton are contained in the set. In addition to a set of allowable behaviors, however, a problem specification must specify the required interface between a solution and its environment. That is, we want a problem specification to be a set of behaviors together with an action signature.

We therefore define a *schedule module* H to consist 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. We denote by $finscheds(H)$ the set of finite schedules of H. The *behavior* of a schedule β of H is the subsequence of β consisting of external actions, and is denoted by $beh(\beta)$. 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 will use the term *module* to refer to either an automaton or a schedule module.

As an example let's consider once more our ABP model. In order to define the ABP specification set, we consider the external actions to be only the *send_msg* and *receive_msg* actions. The problem consists of the set of sequences in which the set of *send_msg* events matches the set of *receive_msg* events and in which an *receive_msg* event must occur after its corresponding *send_msg* event. It is often useful to differentiate between two types of specifications since different techniques are usually used to prove that such specifications are satisfied.

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 (either an automaton or schedule module), let $Behs(M)$, $Finbehs(M)$ and $Fairbehs(M)$ be the schedule modules having the external 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.)

It is convenient to define two operations for schedule modules. Corresponding to our composition operation for automata, we define the composition of a countable collection of strongly compatible schedule modules $\{H_i\}_{i\in I}$ to be the schedule module $H = \prod_{i\in I} H_i$ where:

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

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

Proposition 9 *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 $Fairbehs(A) = \prod_{i\in I} Fairbehs(A_i)$.*

Corresponding to our hiding operation for automata, we define hide $hide_\Sigma H$ to be the schedule module H' obtained from H by replacing $sig(H)$ with $sig(H') = hide_\Sigma sig(H)$.

Finally, we are ready to define a problem specification and what it means for an automaton to satisfy a specification. A *problem* is simply a schedule module P. An automaton A *solves*[1] a problem P if A and P have the same external action signature and $fairbehs(A) \subseteq fairbehs(P)$. An automaton A *implements* a problem P if A and P have the same external action signature (that is, the same external interface) and $finbehs(A) \subseteq finbehs(P)$. Notice that if A solves P, then A cannot be a trivial solution of P since the fact that A is input-enabled ensures that $fairbehs(A)$ contains a response by A to every possible sequence of input actions. For analogous reasons, the same is true if A implements P.

As an example let's consider the ABP specifications that we defined on last page, and let's see how the ABP model can *solve* this problem. We get a solution by hiding all the actions of the composition but the *send_msg* and *receive_msg* actions. Actually, to really prove this, we still should have to argue about the fair executions.

Since we may want to carry out correctness proofs hierarchically in several stages, it is convenient to state the definitions of 'solves' and 'implements' more generally. For example, we may want to prove that one automaton solves a problem by showing that the automaton 'solves' another automaton, which in turn 'solves' another automaton, and so on, until some final automaton solves the original problem. Therefore, let M and M' be modules (either automata or schedule modules) with the same external action signature. We say that *M solves M'* if $fairbehs(M) \subseteq fairbehs(M')$ and that *M implements M'* if $finbehs(M) \subseteq finbehs(M')$.

As we have seen, there are many ways to argue that an automaton A solves a problem P. We now turn our attention to two more general techniques.

[1]This concept is sometimes called *satisfying*.

9.0.7 Proof Techniques

Modular Decomposition

One common technique for reasoning about the behavior of an automaton is *modular decomposition*, in which we reason about the behavior of a composition by reasoning about the behavior of the component automata individually.

It is often the case that an automaton behaves correctly only in the context of certain restrictions on its input. These restrictions may be guaranteed in the context of the composition with other automata comprising the remainder of the system, or may be restrictions defined by a problem statement describing conditions under which a solution is required to behave correctly. A useful notion for discussing such restrictions is that of a module 'preserving' a property of behaviors: as long as the environment does not violate this property, neither does the module.

In practice, this notion is of most interest when the property is prefix-closed, and when the property does not concern the module's internal actions. A set of sequences $\mathcal{P}$ is said to be *prefix-closed* if $\beta \in \mathcal{P}$ whenever both β is a prefix of α and $\alpha \in \mathcal{P}$. A module M (either an automaton or schedule module) is said to be *prefix-closed* provided that $\mathit{finbehs}(M)$ is prefix-closed.

Let M be a prefix-closed module and let $\mathcal{P}$ be a nonempty, prefix-closed set of sequences of actions from a set Φ satisfying $\Phi \cap \mathit{int}(M) = \emptyset$. We say that M *preserves* $\mathcal{P}$ if $\beta\pi|\Phi \in \mathcal{P}$ whenever $\beta|\Phi \in \mathcal{P}$, $\pi \in \mathit{out}(M)$, and $\beta\pi|M \in \mathit{finbehs}(M)$.

In general, if a module preserves a property $\mathcal{P}$, then the module is not the first to violate $\mathcal{P}$: as long as the environment only provides inputs such that the cumulative behavior satisfies $\mathcal{P}$, the module will only perform outputs such that the cumulative behavior satisfies $\mathcal{P}$. This definition, however, deserves closer inspection. First, notice that we consider only sequences β with the property that $\beta\pi|M \in \mathit{finbehs}(M)$. This implies that we consider only sequences β that contain no internal actions of M. Second, notice that we require sequences β to satisfy only $\beta|\Phi \in \mathcal{P}$ rather than the stronger property $\beta \in \mathcal{P}$. Suppose, for example, that $\mathcal{P}$ is a property of the actions Φ at one of two interfaces to the module M. In this case, it may be that for no $\beta \in \mathcal{P}$ and $\pi \in \mathit{out}(M)$ is it the case that $\beta\pi|M \in \mathit{finbehs}(M)$, since all finite behaviors of M containing outputs include activity at both interfaces to M. By considering β satisfying only $\beta|\Phi \in \mathcal{P}$, we consider all sequences determining finite behaviors of M that, at the interface concerning $\mathcal{P}$, do not violate the property $\mathcal{P}$.

One can prove that a composition preserves a property by showing that each of the component automata preserves the property:

Proposition 10 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata and let $A = \prod_{i\in I} A_i$. If A_i preserves $\mathcal{P}$ for every $i \in I$, then A preserves $\mathcal{P}$.*

In fact, we can prove a slightly stronger result. An automaton is said to be *closed* if it has no input actions. In other words, it models a closed system that does not interact with its environment.

Proposition 11 *Let A be a closed automaton. Let $\mathcal{P}$ be a set of sequences over Φ. If A preserves $\mathcal{P}$, then finbehs$(A)|\Phi \subseteq \mathcal{P}$.*

In the special case that Φ is the set of external actions of A, the conclusion of this proposition reduces to the fact that *finbehs*$(A) \subseteq \mathcal{P}$. The proof of the proposition depends on the fact that Φ does not contain any of A's input actions, and hence that if the property $\mathcal{P}$ is violated then it is not an input action of A committing the violation. In fact, this proposition follows as a corollary from the following slightly more general statement: If A preserves $\mathcal{P}$ and $in(A) \cap \Phi = \emptyset$, then *finbehs*$(A)|\Phi \subseteq \mathcal{P}$.

Combining Propositions 10 and 11, we have the following technique for proving that an automaton implements a problem:

Corollary 11.1 *Let $\{A_i\}_{i\in I}$ be a strongly compatible collection of automata with the property that $A = \prod_{i\in I} A_i$ is a closed automaton. Let P be a problem with the external action signature of A. If A_i preserves finbehs(P) for all $i \in I$, then A implements P.*

That is, if we can prove that each component A_i preserves the external behavior required by the problem P, then we will have shown that the composition A preserves the desired external behavior; and since A has no input actions that could be responsible for violating the behavior required by P, it follows that all finite behaviors of A are behaviors of P.

Hierarchical Decomposition

A second common technique for proving that an automaton solves a problem is *hierarchical decomposition* in which we prove that the given automaton solves a second, that the second solves a third, and so on until the final automaton solves the given problem. One way of proving that one automaton A solves another automaton B is to establish a relationship between the states of A and B and use this relationship to argue that the fair behaviors of A are fair behaviors of B. In order to establish such a relationship in between two automata we can use abstraction mappings. But this will be defined on next lecture.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 3: September 18

Lecturer: Nancy A. Lynch *Scribe: David J. Goldstone*

3.1 Liveness

The concept of liveness was introduced in lecture two. Given a set A, the ***alphabet*** of action symbols, we say that L, a set of finite and infinite sequences of A, is a ***liveness property*** provided that every finite sequence has some extension in L. Note that we say that L is a liveness property, rather than L has a liveness property. We think of it as a set of sequences which happen to share an attribute rather than as an attribute of sequences. Sets are easier to deal with than attributes.

An example of the liveness property may be shown for the Alternating Bit Protocol (ABP). The attribute of the liveness property is that the number of receives equals the number of sends. Recall that the attribute of the safety property, S, is that there is consistency between sends and receives. As shown in handout six, there is a correctness property which consists of $S \cap L$.

3.2 ABP Invariants

In making the various correctness arguments about the Input-Output Automata, it is handy to have some invariants around. These invariants are properties which are true for all reachable states (not sequences). They will usually be proven by induction.

In order to state such invariants, we should have an explicit representation of the channels as automata. This is not entirely trivial, because of the liveness requirements. Here I will consider simplified "non-live" versions of the channels; each channel has only one component, a FIFO *queue*, in its state. A *send_pkt*(p) action adds any finite number of copies of p to the end of *queue*, while a *receive_pkt*(p) is enabled when p is the first element on the queue and has the effect of removing this element.[1]

Lemma 3.1 *The following is true about every reachable state of $ABP - S$. Consider the sequence consisting of the indices in queue^{rt} (in order from first to last on the queue), followed by integer^{r}, followed by the indices in queue^{tr}, followed by integer^{t}. (In other words, the*

[1]The live version of the channels will augment this *queue* component with other components that are used to ensure that, in an infinite sequence of *send_pkt* events, infinitely many of the *send_pkt* events result in a nonzero number of packets being placed on *queue*.

sequence starting at $queue^{rt}$, and tracing the indices all the way back to the transmitter automaton.) The indices in this sequence are nondecreasing; furthermore, the difference between the first and last index in this sequence is at most 1.

Proof: The proof proceeds by induction on the number of steps in the finite execution leading to the given reachable state. The base case is where there are no steps, which means we have to show this to be true in the initial state. In the initial state, the channels are empty, $integer^t = 1$ and $integer^r = 0$. Thus, the specified sequence is $0, 1$, which has the required properties.

For the inductive step, suppose that the condition is true in state s', and consider a step (s', π, s) of the algorithm. We consider cases, based on π.

1. π is a *send_msg* or *receive_msg* event. Then none of the components involved in the stated condition is changed by the step, so the condition is true after the step.

2. π is a $send\_pkt^{tr}(m, i)$ event, for some m. Then $queue^{tr}$ is the only one of the four relevant components of the global state that can change. We have $s.queue^{tr}$ equal to $s'.queue^{tr}$ with the addition of a finite number of copies of (m, i). But $i = s'.integer^t$ by the preconditions of the action. Since those new i's are placed consecutively in the concatenated sequence with $s.integer^t = i$, the properties are all preserved.

3. π is a $receive\_pkt^{rt}(i)$ event. If $i \neq s'.integer^t$ then the only change is to remove an element in the concatenated sequence, so all properties are preserved. On the other hand, if $i = s'.integer^t$ then the inductive hypothesis implies that the entire concatenated sequence in state s' must consist of i's. The only changes are to remove one i from the beginning of the sequence and add one $i+1$ to the end (since $s.integer^t = i + 1$, by the effect of the action). Thus, the new sequence consists of all i's followed by one $i + 1$, so the property is satisfied.

4. π is a $send\_pkt^{rt}$ event. Similar to the case for $send\_pkt^{tr}$.

5. π is a $receive\_pkt^{tr}(m, i)$ event. If $i \neq s'.integer^r + 1$ then the only change is to remove an element in the concatenated sequence, so all properties are preserved. On the other hand, if $i = s'.integer^r + 1$ then the inductive hypothesis implies that the entire concatenated sequence in state s' must consist of $i - 1$'s up to and including $s'.integer^r$, followed entirely by i's. The only changes are to change the value of $integer^r$ from $i - 1$ to i, by the effect of the action; this still has the required properties.

■

Lemma 3.2 *The following is true about every reachable state of $ABP - S$. If, in the concatenated sequence, $integer^t$ appears anywhere other than in the transmitter's state, then $buffer^t$ is nonempty.*

Proof: By induction. The only interesting cases are:

1. *send_pkt*^{tr}: easy by precondition

2. *receive_pkt*^{rt}: If the packet is accepted, then the first element gets removed from *buffer*^{t}. But then the inductive hypothesis and the precondition imply that in state s', all the elements of the concatenated sequence are equal. In state s, *integer*^{t} goes up by 1, so is different from all integers elsewhere in the concatenated sequence, and the result is vacuously true.

■

Lemma 3.3 *The following is true about every reachable state of $ABP - S$. All packets in queue^{tr} with integer tag equal to integer^{t} have their message component equal to the first element of buffer^{t}.*

Proof: A similar induction. ■

Thus, we are beginning to accumulate a body of invariants which we will use in further proofs.

3.3 Abstraction Mappings

3.3.1 What are they?

Abstraction mappings are functions which we use in order to depend on automata we understand to describe automata we do not understand.

Definition Suppose A and B are input/output automata with the same external action signature, and suppose f is a mapping from *states*(A) to *states*(B). f is an **abstraction mapping** from A to B provided:

1. If $s_0 \in \text{start}(A)$ then $f(s_0) \in \text{start}(B)$; and

2. If s' is a reachable state of A, and $f(s')$ is a reachable state of B, and (s', π, s) is a step of A, then there is an 'extended step' $(f(s'), \gamma, f(s))$ such that $\gamma|\text{ext}(B) = \pi|\text{ext}(A)$.

An extended step of an automaton A is a triple of the form (s', β, s), where s' and s are states of A, β is a finite sequence of actions of A, and there is an execution fragment of A having s' as its first state, s as its last state, and β as its schedule. The following theorem is the core of this paragraph:

Theorem 3.4 *If there is an abstraction mapping from A to B, then behs$(A) \subseteq$ behs(B).*

The proof of this theorem is asked in homework 2.

3.3.2 An example

Now we will show that $ABP - S$ satisfies the safety property given by the specification: that the sequence of messages occurring in *receive_pkt* events is consistent with the sequence of messages occurring in *send_pkt* events. A nice way to do this is by showing a mapping to another automaton, MQ representing a message queue. MQ has the signature:

Inputs:
  send_msg(m)
Outputs:
  receive_msg(m)

The state has one component, *mqueue*. The *send_msg*(m) action simply adds a single copy of m to the end of *mqueue*, while the *receive_msg*(m) is enabled when m is first on *mqueue* and removes the message.

We prove a (single-valued) mapping from $ABP - S$ to MQ. More precisely, we define the mapping f from states of $ABP - S$ to states of MQ. Given a reachable state s of $ABP - S$ (which includes states of the transmitter, receiver and both channels), if $s.integer^t = s.integer^r$, then the *mqueue* in state $f(s)$ is constructed by first removing the first element of $buffer^t$ and then concatenating $buffer^r$ and the "reduced" $buffer^t$, in that order. (Note: Lemma 3.2 implies that $s.buffer^t$ is nonempty.) Otherwise, the *mqueue* is just the concatenation of $buffer^r$ and $buffer^t$. (We can define the mapping arbitrarily for non-reachable states; it doesn't matter anyway.)

Now we show that f is an abstraction mapping, as defined in class. The correspondence of initial states is easy because all queues are empty.

Inductive step: Given (s', π, s), and $u' = f(s')$, we consider the following cases.

1. *send_msg*(m): Let $u = f(s)$. We must show that (u', π, u) is a step of MQ. This is true because the *send_msg* event modifies both queues in the same way.

2. *receive_msg*(m): Let $u = f(s)$. We must show (u', π, u) is a step of MQ. Since π is enabled in s', m is first on $s'.buffer^r$, so m is also first on $u'.buffer^r$, so π is also enabled in u'. Both queues are modified in the same way.

3. *send_pkt*: Then we must show that $u' = f(s)$. This is true because the action doesn't change the virtual queue.

4. *receive_pkt*$^{tr}(m, i)$: We show that $u' = f(s)$. The only case of interest is if the packet is accepted. This means $i = s'.integer^r + 1$, so in state s', the two integers are different, so $f(s') = u'$ is the concatenation of $s'.buffer^r$ and $s'.buffer^t$.

   Then in s, the integers are equal, and m is added to the end of $buffer^r$. So $f(s)$ is determined by removing the first element of $s.buffer^t$ and then concatenating with

$s.buffer^r$. Then in order to see that $f(s)$ is the same as u', it suffices to note that m is the same as the first element in $s'.buffer^t$. But Invariant 3.1 implies that $i = s'.integer^t$, and then Invariant 3.3 implies that m is the same as the first element in $s'.buffer^t$, as needed.

5. $receive\_pkt^{rt}(i)$: Again we show that $u' = f(s)$. The only case of interest is if the packet is accepted. This means $i = s'.integer^t$, which implies that $s'.integer^t = s'.integer^r$. This means that $f(s')$ is the concatenation of $s'.buffer^r$ and $s'.buffer^t$, with the first element of $s'.buffer^t$ removed. Then $s.integer^t = s.integer^r + 1$, so $f(s)$ is just the concatenation of $s.buffer^t$ and $s.buffer^r$. But $s.buffer^t$ is obtained from $s'.buffer^t$ by removing the first element, so that $f(s) = u'$.

Since f is an abstraction mapping, the theorem implies that all behaviors of $ABP - S$ are also behaviors of MQ, and so satisfy the safety property that the sequence of messages in *receive_msg* events is consistent with the sequence of messages in *send_msg* events. In particular, the fair behaviors of $ABP - S$ satisfy this safety property.

3.4 Introduction to Multivalued Mappings

Multivalued mappings are similar to abstraction mappings. In essence, a multivalued mapping f is a function which maps a state s of a machine A which we don't understand completely to a set of states $f(s)$ of a machine B we do understand completely. Typically, B is an "inefficient" (in the sense that its state space is big), but conceptually simple algorithm, whereas A has an "optimized" state space. A good example of the directionality inherent in these mappings is garbage collection, because states exist in the un-garbage collected machine (B) which don't exist in the clean machine (A). Another example we saw in class is $B =$ ABP-S, $A =$ ABP.

6.852 Distributed Algorithms	**Fall Semester, 1990**
Lecture 4: September 20	
Lecturer: Nancy A. Lynch	*Scribe: Mary Ellen Zurko*[2]

4.1 Possibilities Mapping

Definition Suppose A and B are I/O Automata (IOA) with the same external action signature, and suppose f is a mapping from states(A) to the power set of states(B). Then f is a *possibilities mapping* provided that:

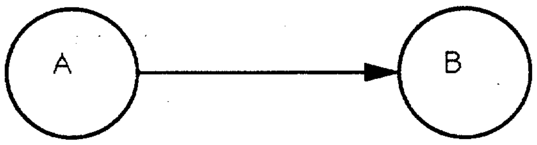


Figure 4.1: A mapping from A to B via f

1. If $s_0 \in start(A)$ then $\exists$ start state t_0 of B s.t. $t_0 \in f(s_0)$
2. If s' is a reachable state of A, $t' \in f(s')$ is a reachable state of B, and if (s', π, s) is a step of A then there is an extended step (t', γ, t) of B s.t. $\gamma | ext(B) = \pi | ext(A)$ and $t \in f(s)$.

Note that states(B) can be infinite even though states(A) is finite; also, for two distinct elements x and x' of A, we do not exclude the fact that $f(x)$ and $f(x')$ might intersect.

Theorem 4.1 *If there is a possibilities mapping from A to B, then* $behs(A) \subseteq behs(B)$.

Proof: Let $\pi \in behs(A), \pi = \pi_1 \pi_2 \ldots$ Call $P_k(\pi) = \pi_1 \pi_2 \ldots \pi_k$.

- We prove by induction on k that there is an execution e'_k of B whose behavior is $P_k(\pi)$:

  <u>$k = 0$</u>: This is part 1 of the definition.

  <u>Assume this is true for k.</u> Hence we have found an execution e'_k corresponding to $P_k(\pi)$. Let t'_k be the final state of B after e'_k. Let $(t'_k, \gamma_{k+1}, t_{k+1})$ be the extended step of B

[2]Based on lecture notes from 1988 scribed by Chris Colby.

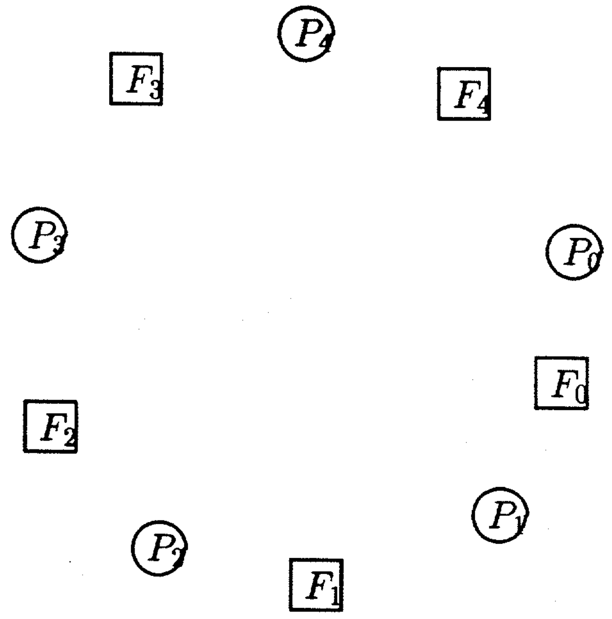


Figure 10.1: Dining Philosophers problem (n = 5)

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

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 packet in C^{tr} in u, then $(m, i\bmod 2)$ is the j^{th} packet in C^{tr} in s. Also, C^{rt} has the same number of packets in s and u. Moreover, for any j, if i is the j^{th} packet in C^{rt} in u, then $i\bmod 2$ is the j^{th} packet in C^{rt} in s.

Theorem 4.2 *g above is a possibilities mapping.*

Proof: By induction. For the base, let s be the start state of ABP and u the start state of $ABP - S$. First, all the buffers are empty, which suffices. Second, $s.\mathit{flag}^t = 1 = u.\mathit{integer}^t \bmod 2$ and $s.\mathit{flag}^r = 0$ and $u.\mathit{integer}^r = 0$, which is as needed. Third, both channels are empty.

Now show the inductive step. Suppose (s', π, s) is a step of ABP and $u' \in g(s')$. We consider cases based on π. .

1. $\pi = \mathit{send\_msg}(m)$

   Choose u to be the unique state such that (u', π, u) is a step of $ABP - S$. We must show that $u \in g(s)$. The only condition that is affected by the step is the first, for the buffer^t component. However, the action affects both $s.\mathit{buffer}^t$ and $u.\mathit{buffer}^t$ in the same way, so the correspondence holds.

2. $\pi = \mathit{receive\_msg}(m)$

   Since π is enabled in s', m is the first value on $s'.\mathit{buffer}^r$. Since $u' \in g(s')$, m is also the first value on $u'.\mathit{buffer}^r$, which implies that π is enabled in u'. Now choose u to be the unique state such that (u', π, u) is a step of $ABP - S$. All conditions are unaffected except for the first for buffer^r, and buffer^r is changed in the same way in both algorithms, so the correspondence holds.

3. $\pi = \mathit{send\_pkt}^{tr}(m, b)$

   Since π is enabled in s', $b = s'.\mathit{flag}^t$ and m is the first element on $s'.\mathit{buffer}^t$. Let $i = u'.\mathit{integer}^t$. Since $u' \in g(s')$, m is also the first element on $u'.\mathit{buffer}^t$. It follows that $\bar{\pi} = \mathit{send\_pkt}^{tr}(m, i)$ is enabled in u'.

   Now choose u so that $(u', \bar{\pi}, u)$ is a step of $ABP - S$ and such that this step puts the same number of packets in C^{tr} as does the step (s', π, s). We must show that $u \in g(s)$. The only interesting condition is the third, for C^{tr}. By inductive hypothesis and the fact that the two steps insert the same number of packets, it is easy to see that C^{tr} has the same number of packets in s and u. Moreover, the new packets get added with tag i in state u and with tag b in state s; since $u' \in g(s')$, we have $s'.\mathit{flag}^t = u'.\mathit{integer}^t \bmod 2$, i.e., $b = i\bmod 2$, which implies the result.

4. $\pi = \mathit{receive\_pkt}^{tr}(m, b)$

Since π is enabled in s', (m, b) is the first element in C^{tr} in s'. Since $u' \in g(s')$, (m, i) is the first element in C^{tr} in u', for some integer i with $b = i \bmod 2$. Let $\bar{\pi} = \mathit{receive\_pkt}^{tr}(m, i)$; then $\bar{\pi}$ is enabled in u'. Let u be the unique state such that $(u', \bar{\pi}, u)$ is a step of $ABP - S$. We must show that $u \in g(s)$.

It is easy to see that the third condition is preserved, since each of π and $\bar{\pi}$ simply removes the first packet from C^{tr}.

Suppose first that $b = s'.\mathit{flag}^r$. Then the effects of π imply that the receiver state in s is identical to that in s'. Now, since $u' \in g(s')$, $s'.\mathit{flag}^r = u'.\mathit{integer}^r \bmod 2$; since $b = i \bmod 2$, this case must have $i \neq u'.\mathit{integer}^r + 1$. Then the effects of $\bar{\pi}$ imply that the receiver state in u is identical to that in u'. It is immediate that the first and second conditions hold.

So now suppose that $b \neq s'.\mathit{flag}^r$. The invariant above for $ABP - S$ implies that either $i = u'.\mathit{integer}^r$ or $i = u'.\mathit{integer}^r + 1$. Since $b = i \bmod 2$ and (since $u' \in g(s')$) $s'.\mathit{flag}^r = u'.\mathit{integer}^r \bmod 2$, this case must have $i = u'.\mathit{integer}^r + 1$. Then by the effect of the action, $u.\mathit{integer}^r = u'.\mathit{integer}^r + 1$ and $s.\mathit{flag}^r = 1 - s'.\mathit{flag}^r$, preserving the second condition. Also, buffer^r is modified in both cases by adding the entry m at the end; therefore, the first condition is preserved.

5. $\pi = \mathit{send\_pkt}^{rt}(b)$

   Similar to $\mathit{send\_pkt}^{tr}(m, b)$.

6. $\pi = \mathit{receive\_pkt}^{rt}(b)$

   Since π is enabled in s', b is the first element in C^{rt} in s'. Since $u' \in g(s')$, i is the first element in C^{rt} in u', for some integer i with $b = i \bmod 2$. Let $\bar{\pi} = \mathit{receive\_pkt}^{rt}(i)$; then $\bar{\pi}$ is enabled in u'. Let u be the unique state such that $(u', \bar{\pi}, u)$ is a step of $ABP - S$. We must show that $u \in g(s)$.

   It is easy to see that the third condition is preserved, since each of π and $\bar{\pi}$ simply removes the first message from C^{rt}.

   Suppose first that $b \neq s'.\mathit{flag}^t$. Then the effects of π imply that the transmitter state in s is identical to that in s'. Now, since $u' \in g(s')$, $s'.\mathit{flag}^t = u'.\mathit{integer}^t \bmod 2$; since $b = i \bmod 2$, this case must have $i \neq u'.\mathit{integer}^t$. Then the effects of $\bar{\pi}$ imply that the transmitter state in u is identical to that in u'. It is immediate that the first and second conditions hold for this situation.

   So now suppose that $b = s'.\mathit{flag}^t$. The invariant above for $ABP - S$ implies that either $i = u'.\mathit{integer}^t - 1$ or $i = u'.\mathit{integer}^t$. Since $b = i \bmod 2$ and (since $u' \in g(s')$) $s'.\mathit{flag}^t = u'.\mathit{integer}^t \bmod 2$, this case must have $i = u'.\mathit{integer}^t$. Then the effect of the action implies that $u.\mathit{integer}^t = u'.\mathit{integer}^t + 1$ and $s.\mathit{flag}^t = 1 - s'.\mathit{flag}^t$, preserving the second condition. Also, *buffer*^{t} is modified in the same way in both cases, so the first condition is preserved.

■

Remarks

Consider the structure of the possibilities mapping g of this example. In going from $ABP-S$ to ABP, integer tags are condensed to their low-order bits. The multiple values of the mapping g essentially "replace" this information. In this example, the correspondence between ABP and $ABP-S$ can be described in terms of a mapping in the opposite direction - a (single-valued) projection from the state of $ABP-S$ to that of ABP that removes information. Then g maps a state s of ABP to the set of states of $ABP-S$ whose projections are equal to s. While this formulation suffices to describe many interesting examples, it does not always work. (Consider garbage collection examples.)

Here we were able to exhibit a possibilities mapping and prove it correct. The practical problem this method generates is that, if we make a mistake in the definition of f and discover this mistake while working on a particular step of the proof, we then need to fix f and might have to consider re-checking all cases that we previously checked on the wrong version. Machine assistance can be useful here.

4.1.2 Liveness of ABP

Techniques for showing liveness are more ad hoc. Here is a sketch of *one* proof that works for showing that the fair behaviors of ABP all satisfy the main correctness condition: that the subsequence of messages received is identical to the subsequence of messages sent.

We want to show that each fair behavior of ABP satisfies that: # messages sent = # messages received

It suffices for this to show that every *fairbeh* of ABP is also a *fairbeh* of MQ (since they all satisfy the condition).

First, we can get a (single-valued) possibilities mapping h from ABP directly to MQ; this is constructed in much the same way as f above, only the test that determines whether the concatenation occurs is based on whether or not the *flag*^{t} and *flag*^{r} values are equal. (If they are equal, the first element of *buffer*^{t} is removed before concatenation with *buffer*^{r}.)

It is sufficient to show the following. For any fair execution α of ABP, the "image", α', of this execution under h is a fair execution of MQ. (This image involves mapping each state to its image under h, and removing occurrences of actions of ABP that are not actions of MQ.) Saying that α' is a fair execution of MQ implies that any sent message must eventually be received.

Suppose for the purpose of contradiction that this is not the case, so α' is not fair. This means that, in α', there is a *receive_msg* action enabled from some point i on, but no *receive_msg* occurs after that point. Now, whenever a *receive_msg* action is enabled in MQ, it means that *mqueue* is nonempty. By the operation of MQ, this means that a particular message m appears first on *mqueue* in the state at point i of α', and that it persists forever afterwards in α'.

Now consider α. By the definition of mapping h, from point i onward in α it must be that the concatenation of $buffer^t$ and $buffer^r$ (with the first element of $buffer^t$ removed if the two flags are equal) is a queue that has m in the first position. More specifically, in each state after point i, one of the following holds.

1. $buffer^r$ is nonempty,
2. $buffer^r$ is empty, $flag^t \neq flag^r$ and m is first on $buffer^t$, or
3. $buffer^r$ is empty, $flag^t = flag^r$ and m is second on $buffer^t$.

First we claim that no state in this interval has condition 1 true. For if it did, then this condition must continue to hold forever thereafter until and unless a *receive_msg* event occurs; this means that a *receive_msg* event is enabled in all states from that point on. But then the fairness definition implies that a *receive_msg* must eventually occur. This is a contradiction. So condition 1 can never hold.

Next we claim that no state in this interval can have condition 2 true. For if it did, it must continue to hold until and unless a $receive\_pkt^{tr}(*, flag^t)$ event occurs. (It is impossible to have a $receive\_pkt^{rt}$ event falsify this condition, because a version of Lemma 1.1 in Section 1 of Handout 7, stated for ABP rather than $ABP-S$, implies that, when this condition is true, all tags in C^{rt} must be unequal to $flag^t$ and so will not be accepted by the transmitter. The invariant could be carried over to ABP using the mapping g above.) If a $receive\_pkt^{tr}(*, flag^t)$ event occurs, then $buffer^r$ becomes nonempty, which means that condition 1 holds. Since we have already proved 1 can't hold, this means that no $receive\_pkt^{tr}(*, flag^t)$ event can occur. But note that $send\_pkt^{tr}(m, flag^t)$ continues to occur infinitely often, by the fairness definition (since it remains enabled). This means that infinitely many $receive\_pkt^{tr}$ events must occur, by the channel liveness assumption. Eventually, the finitely many packets in C^{tr} with tag different from $flag^t$ will have been received, and so there must eventually be some with tag equal to $flag^t$ received. This is a contradiction. The conclusion is that condition 2. can never hold.

So it must be that condition 3 holds everywhere after point i. This means that no event of the form $receive\_pkt^{rt}(*, flag^t)$ can ever occur in that interval (since that would make condition 2 hold). But infinitely many $send\_pkt^{rt}(flag^t)$ events occur in this interval, by fairness. By the channel liveness assumption, an event of the form $receive\_pkt^{rt}(*, flag^t)$ must eventually occur, a contradiction.

Note that this proof could be done more "formally", using, e.g., temporal logic. But arguments in English can also be done rigorously.

Unfortunately, there are no nice systematic methods to apply. Some examples of methods in the literature are given below:

- Variant Functions. They are not used in Handout 7. Many papers use it in the literature. The idea of this method is the following:

  We construct a function $states(A) \xrightarrow{f} E$, where E is a well founded set (by this we mean that E has an order $\leq$, for which there is no infinite strictly decreasing chain). The function f is assumed to be such that if (s, π, s') is a step of A then $f(s') \leq f(s)$. We assume also that f is strictly decreasing in the following sense: in any state, some class is guaranteed to (strictly) decrease the value of f when it next executes. Call E' the subset of E consisting of all the minimal elements of E. We assume that for any execution e ending with the state s', if $f(s') \in E'$ then e' is an execution verifying our liveness condition.

  In this method, the function f measures in some sense "how far an execution is from the liveness goal".

- Temporal Logic. It uses special symbols for "something eventually happens": $\diamond\pi$.

- Unity language. It has a set of proof rules for the concept "leads-to".

None of these methods have proven general enough for all interesting cases.

Of course, we can always argue directly as we did with ABP about the execution sequence and show that every fair behavior has the liveness property.

4.2 Mutual Exclusion Using Shared Memory

4.2.1 Introduction

The problem of Mutual Exclusion is the problem of the allocation of an indivisible resource among any subset of n processes $p_1, \ldots, p_n$. Mutual exclusion Algorithms are distributed Algorithms solving this problem in such a way that each process p_i holds a copy $\mathcal{C}_i$ of the same code $\mathcal{C}$. The code $\mathcal{C}$ is partitioned into four sections: the *remainder region* (R), the *trying region* (T), the *critical region* (C), or the *exit region* (E). We say that process i is, at some point, in region R,T,C or E if the line of its code $\mathcal{C}$ it is currently executing is in R,T,C or E. Theses four regions are defined by the following facts. A process not engaged in the competition for the resource is in R. When a process is trying to gain control of the resource, it is in T. When it holds the resource, it is in C. Eventually, when it is done with the resource, it goes to E where it typically reinitializes some variables, and then goes back to R. The cycle of regions that a process visits is shown in Figure 4.3. We will consider here only algorithms in which communication among processes is done through shared variables.

$$\begin{tikzcd} R \arrow[r] & T \arrow[r] \arrow[loop above] & C \arrow[r] & E \arrow[lll, bend left] \end{tikzcd}$$

Figure 4.3: The cycle of regions of a single process.

Within this setting mutual exclusion algorithms are characterized by the two following properties:

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

"Normal operation" means that any process in regions T or E is provided with a step by the scheduler and that furthermore, a process in region C is eventually asked to leave (to region E) by the user. "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 modeled 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 ($user_1, \ldots, user_n$). 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 $user_i$. Figure 4.4 illustrates this series of actions.

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

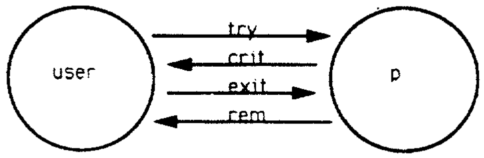


Figure 4.4: Input/Output Actions in between p_i and user$_i$

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 4.5.

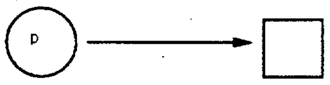


Figure 4.5: p_i reads v and stores the value in its local variable

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

Modeling via I/O automaton

We will consider the algorithm as being modeled as *one* big I/O automaton as illustrated in Figure 4.6.

The *state* of the I/O holds the states of all the individual processes, along with the values held by the shared variables. The state of each individual process p_i consists of the values of all the process' local variables along with the value held by its program counter.

The *initial state* of the I/O automaton contains the initial values for all the local variables, and for the shared variables. The program counters are all at the beginning (the initial region of all processes is R).

The *actions* are T, C, E, R, access to variables, and local computations. The action Try$_i$ and Exit$_i$ are the inputs actions of the I/O automaton.

The *partitions.* The actions of each p_i are in one class.

The *steps* follow the code in a natural way. The code doesn't explicitly mention Try, Crit, Exit, or Rem actions; these are implicit. When a Try occurs, the program counter moves to the beginning of the trying region code. The process is then enabled to perform steps and progresses through the trying region code in the usual way. When it is done with the Trying

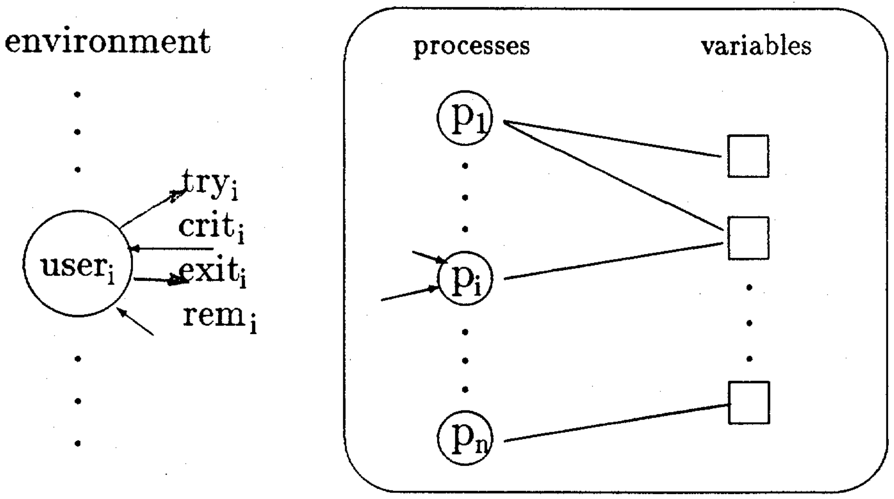


Figure 4.6: A global view of the mutual exclusion problem.

protocol, a Crit action occurs, thereby setting the program counter after the Try code. No actions are then enabled until an input action occurs Exit. We reason analogously for E and R. Recall also that by definition fair behaviors, processes keep taking steps as long as they are enabled.

The liveness properties of this problem can be reexpressed as: If the environment eventually does an exit$_i$ action when p_i is in C, then progress must always eventually occur. (Assume that processors continue to take steps in T and E if they are "enabled").

We could model this code by an I/O automaton in another way: each process and each shared variable could be a separate I/O automaton, interacting via composition. This adds more complexity though, since the shared variable I/O automaton's have separate actions for invocation and response.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 5: September 25

Lecturer: Nancy Lynch *Scribe: Mary Ellen Zurko,Ken DeCanio*[3]

5.1 Dijkstra's algorithm

In 1965, the first mutual exclusion algorithm was developed by Edsgar W. Dijkstra. It is presented in Figure 5.1 as the code to be run on each process. The language used in the description of this code is different from the precondition-effect language used in previous lectures, but the translation can easily be made.

Region T runs from the label L till the comment "*critical section*". The algorithm uses two variables *Control*[i] and k. *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, so that we need to define what are the *atomic action.* For this algorithm, atomic actions are reads from and writes to the shared variables. These are *read-write* registers. The code is rewritten in Figure 5.2 so as to put in clear evidence the atomic actions: these are enclosed in pointy brackets. (We review this code in detail in in the sequel. The sets S_i will be defined before Theorem 5.4.) 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.

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 5.1 *Dijkstra's algorithm preserves the cyclicity Try, Crit, Exit, Rem of the actions of user*$_i$.

[3]Based on lecture notes from 1988 scribed by Sharon Pearl

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: $control[i] \leftarrow 1$
while $k \neq i$ do
    if $control[k] = 0$ then $k \leftarrow i$
    end if
end while

  **Begin 2nd stage, checking that no other processor has reached this stage.**

$control[i] \leftarrow 2$
for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$ do **Note: order of checks unimportant**
    if $control[j] = 2$ then goto L
    end if
end for

  **Critical region**

$control[i] \leftarrow 0$

  **Remainder region**
```

Figure 5.1: 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: $S_i \leftarrow \emptyset$
$\langle control[i] \leftarrow 1 \rangle$
while $\langle k \neq i \rangle$ do
    if $\langle control[k] = 0 \rangle$ then $\langle k \leftarrow i \rangle$
    end if
end while
 **Begin 2nd stage, checking that no other processor has reached this stage.**
$\langle control[i] \leftarrow 2 \rangle$
for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$ do **Note: order of checks unimportant**
    if $\langle control[j] = 2 \rangle$ then goto L
    end if
    $S_i \leftarrow S \cup \{j\}$
end for
 **Critical region**
$S_i \leftarrow \emptyset$
$\langle control[i] \leftarrow 0 \rangle$
 **Remainder region**
```

Figure 5.2: Dijkstra's algorithm showing atomic actions and S_i.

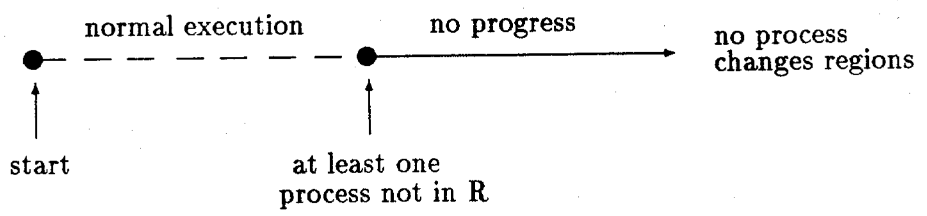


Figure 5.3: No progress.

Proof: Clear. ∎

Theorem 5.2 *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 occurred 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 5.3 *Dijkstra's algorithm satisfies 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 5.3). 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* ≥ 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 contender's index, $\mathit{control}[k] \geq 1$. Then, any future reads of k and $\mathit{control}[k]$ will yield $\mathit{control}[k] \geq 1$, 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 $\mathit{control}[k]$ is set to 2. The only way that p_k could fail to reach C immediately is if $\mathit{control}[i] = 2$, 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.

This sounds somewhat informal. An argument about finite execution sequences can be made rigorous, but you must be quite careful when arguing temporal concepts like "before", "until", etc. Such proofs are error-prone because they are quite complex. ∎

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* or *invariant assertions.* 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 $\mathit{control}[j] \neq 2$ (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 $\mathit{control}[i]$ is set to 0 or 1, $S_i \leftarrow \emptyset$. Figure 5.2 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 |in-C| ≤ 1. However, the stronger statement |in-C|+|before-C| ≤ 1 is sufficient. The following two claims imply that statement:

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

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 5.4 *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 5.5 *In any state reachable during an execution of Dijkstra's algorithm |in-C| + |before-C| ≤ 1.*

It should be obvious that Theorem 5.4 is a corollary of Theorem 5.5. In order to prove Theorem 5.5 we will need a small set of lemmas and definitions.

First a small lemma.

Lemma 5.6 *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 5.5 says, in effect, that two processes in stage 2 cannot both miss each other's stage 2 control signals.

Lemma 5.7 $\nexists p_i, p_j$ $[i \neq j$ and $i \in S_j$ 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 5.6 then, $S_j = \emptyset$, and so $i \notin S_j$. ∎

The second lemma that we will use to prove Theorem 5.5 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 5.8 $p_i \in (\textit{before-C} \cup \textit{in-C}) \Rightarrow S_i = \{1, \ldots, n\} - \{i\}$.

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 5.5 by contradiction:

Proof: Assume that in some state reachable in an execution $|\textit{in-C}| + |\textit{before-C}| > 1$. Then there exist two processes, p_i and p_j, $i \neq j$, such that $p_i \in (\textit{before-C} \cup \textit{in-C})$ and $p_j \in (\textit{before-C} \cup \textit{in-C})$. By Lemma 5.8, $S_i = \{1, \ldots, n\} - \{i\}$ and $S_j = \{1, \ldots, n\} - \{j\}$. But by Lemma 5.7, 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. ■

See Goldman-Lynch for a detailed proof.

This concludes our examination of Dijkstra's algorithm.

5.2 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 additional mutual exclusion algorithms that improve upon Dijkstra's algorithm in different ways, and one impossibility result about mutual exclusion.

5.2.1 Eisenberg-McGuire Mutual Exclusion Algorithm

The Eisenberg-McGuire Mutual Exclusion Algorithm (see Figure 5.4) 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.)

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.

5.2.2 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 distributed systems), algorithms that use only single-writer variables are worth investigating.

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: $control[i] \leftarrow 1$
for $j = k, k+1, \ldots, n, 1, \ldots, k-1$ do
    if $j = i$ then exit
    end if
    if $control[j] \geq 1$ then goto L
    end if
end for

  **Begin 2nd stage, identical to Dijkstra's algorithm.**

$control[i] \leftarrow 2$
for $j \in \{1, \ldots, i-1, i+1, \ldots, n\}$ do
    if $control[j] = 2$ then goto L
    end if
end for

  **Begin 3rd stage, making final check.**

if $control[k] \geq 1$ and $k \neq i$ then goto L
end if
$k \leftarrow i$

  **Critical region**

  **Select as successor the next contending process.**

for $j = k+1, \ldots, n, 1, \ldots, k-1$ do
    if $control[j] \neq 0$ then
        $k \leftarrow j$
        exit
    end if
end for
$control[i] \leftarrow 0$

  **Remainder region**
```

Figure 5.4: Eisenberg-McGuire Mutual Exclusion Algorithm

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: $control[i] \leftarrow 0$
for $j \in \{1,\ldots,i-1\}$ do
    if $control[j] = 1$ then goto L
    end if
end for
$control[i] \leftarrow 1$
for $j \in \{1,\ldots,i-1\}$ do
    if $control[j] = 1$ then goto L
    end if
end for
M:
for $j \in \{i+1,\ldots,n\}$ do
    if $control[j] = 1$ then goto M
    end if
end for

  **Critical region**

$control[i] \leftarrow 0$

  **Remainder region**
```

Figure 5.5: Burns' Mutual Exclusion Algorithm

The next algorithm, developed by Jim Burns, appears in Figure 5.5. 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 5.9 *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 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 in order 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 important aspects of the model are the specification of "normal operation", the definition of actions along with the specifications of the components that control them.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 6: September 27

Lecturer: Nancy A. Lynch *Scribe: Andrew Chou*[4]

In this lecture, we will look at two new algorithms: Lamport's Mutual Exclusion algorithm and the Peterson-Fisher algorithm. Previously, we have covered:

- Dijkstra's algorithm - Mutual exclusion and progress (multi-writer variables).
- Eisenburg-McGuire algorithm - Mutual exclusion, progress, and no lockout (multi-writer variables).
- Burn's algorithm - Mutual exclusion, progress, single-writer variables, but with lockout.

Both new algorithms are single-writer, no lockout algorithms. Lamport's algorithm uses unbounded variables, while the Peterson-Fischer algorithm uses binary valued variables.

6.1 Lamport's "Bakery" Mutual Exclusion Algorithm

Lamport's "Bakery" algorithm (see Figure 6.1) 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.* (We will define these registers in the next section.)
- It satisfies fairness: the first part of the trying region, the "doorway", is wait-free, and the second part, the "bakery", insures FIFO priority.
- It achieves a degree of *failure resiliency.*

[4]Based on lecture notes from 1988 scribed by Steven Ponzio

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 $\{0,1,\ldots\}$, initially all 0, where $number[i]$ is written by p_i and read by all

Code for p_i

```
 ** beginning of doorway **
L1: $choosing[i] \leftarrow 1$
$number[i] \leftarrow 1 + max(number[1], \ldots, number[n])$
$choosing[i] \leftarrow 0$

 ** end of doorway **

 ** beginning of bakery**
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**

 ** end of bakery **
$number[i] \leftarrow 0$

 **Remainder region**
```

Figure 6.1: Lamport's "Bakery" Mutual Exclusion Algorithm

6.1.1 Atomic and Safe Registers

I/O Automaton Modeling of a Register

The entity controlling the access to a variable is modeled as an I/O automaton $\mathcal{R}$. Denote n as the number of processes. Each process trying to access the variable produces an input action to $\mathcal{R}$. To each *read* input action corresponds a subsequent output action *read−value*, and to each *write* input action corresponds a subsequent output action *write − ack*. Each input action happens at some time t_{req}. The corresponding output action happens at some time t_{ack} ($t_{ack} \geq t_{req}$). An atomic (resp. safe) variable is a variable whose sequence of times t_{ack}, t_{req} verify specific properties that we make precise in the sequel. 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.)

Atomic Variables

We begin with an informal description. A variable is 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} ($t_{req} \leq t_{acc} \leq t_{ack}$), such that the entire access could be thought of as happening at t_{acc}. (see Figure 6.2.)

For any execution $\mathcal{E}$ (resp. behavior α) and for any $j = 1 \ldots n$, call $\Pi_j(\mathcal{E})$ (resp. $\Pi_j(\alpha)$) the execution (resp. behavior) obtained by considering only actions involving the jth process. Hence an action of $\mathcal{E}$ is kept in $\Pi_j(\mathcal{E})$ only if it is an internal action of j, or if is an input action to j or an output action from j. For any admissible execution $\mathcal{E}$ of $\mathcal{R}$, let $t_{req,1}$, $t_{req,2}$, ... denote times at which input actions happen. Let $t_{ack,1}$, $t_{ack,2}$, ... denote the times at which corresponding output actions happen. (By convention, $i \mapsto t_{req,i}$ is increasing. But, $i \mapsto t_{ack,i}$ need not be.) Denote α the behavior of $\mathcal{E}$. The variable $\mathcal{R}$ is atomic if, for all admissible executions $\mathcal{E}$ of $\mathcal{R}$, there exists an admissible execution $\mathcal{E}'$ (we let α' denote the behavior of $\mathcal{E}'$), such that for all i, there exists a time $t_{acc,i} \in [t_{req,i}, t_{ack,i}]$ verifying the two following properties:

1. $t'_{req,i} = t'_{ack,i} = t_{acc,i}$, (the times t' refer to the execution $\mathcal{E}'$),
2. for all $j = 1, \ldots, n : \Pi_j(\alpha') = \Pi_j(\alpha)$.

Condition (1) expresses that the "interval executions are shrunk to points". Condition (2) expresses that the two executions $\mathcal{E}$ and $\mathcal{E}'$ are "consistent".

Safe Registers

It is possible to define a weaker kind of register called the safe register. It is weaker then the atomic register in the sense that any atomic register is a safe register. Assuming that

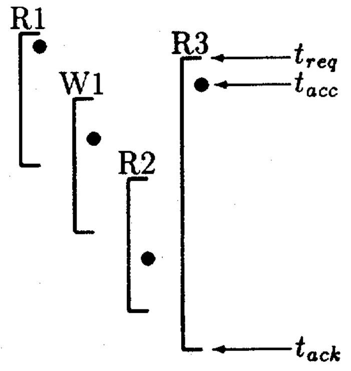


Figure 6.2: Atomic variables: Each bracket represents a read or write cycle, with the vertical dimension representing time. Variables are *atomic* if there exists some set of times t_{acc} such that for each cycle's t_{acc}, the entire cycle could be conceptually "shrunk" to that point in time without changing the values of any responses.

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. Lamport's Bakery algorithm only requires safe registers.

6.1.2 Overview of Lamport's Algorithm

Lamport's algorithm guarantees fairness by using a partially FIFO trying region. The trying region $\mathcal{T}$ is broken up into two sub-regions: $\mathcal{T}_1$ 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. It is possible for two processes to enter and leave "the doorway," concurrently. These processes will, thus, have the same *number*. We then break the tie by comparing process indices (The tie-breaking decision is arbitrary). $\mathcal{T}_2$ 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.) This behavior makes the doorway wait-free.

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.

6.1.3 Properties

The arguments presented here are a little trickier, because of safe registers. Instead of atomic events for accessing registers, we must consider pairs of atomic events for the beginning and ending of each access. This makes assertional proofs difficult, due to the large number of states. Therefore, we will give operational proofs.

Claim 6.1 *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 6.2 *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 6.1, we have $(number[i], i) < (number[j], j)$ and $(number[i], i) > (number[j], j)$ simultaneously, which is impossible. ■

Claim 6.3 *Progress is satisfied.*

Proof: All processes in $\mathcal{T}_1$ enter eventually $\mathcal{T}_2$. In $\mathcal{T}_2$, the process with the lowest pair $(number[i], i)$ may proceed unhindered. ■

Claim 6.4 *Lockout is not possible.*

Proof: Once a process p_i enters the $\mathcal{T}_2$, it has priority over all the processes p_j that are not in $\mathcal{T}_2$ at its time of entry. Only those processes that were in $\mathcal{T}_2$ before p_i can go to $\mathcal{C}$ before p_i. As progress is satisfied, it will also eventually enter into $\mathcal{C}$. ■

Lamport's algorithm can also handle limited failures. Specifically, failures (not in the critical region) that put the process in the remainder region with initialized parameters are allowed. These types of failures do not affect mutual exclusion or fairness. However, if a failed process is allowed to restart, progress is no longer guaranteed. Consider the following situation: p_1 enters the trying region and reaches L2. p_2 enters the trying region, and $choosing[2] \leftarrow 1$. At this point, the scheduler allows p_1 to act. $choosing[2] = 1$, so p_1 loops at L2. p_2 now fails ($choosing[2] \leftarrow 0$), restarts, and enters the trying region again ($choosing[2] \leftarrow 1$). Once again, p_1 is allowed to act and loops at L2. p_2 fails, restarts, and enters $\mathcal{T}$. This procedure can repeat itself, infinitely often, and progress will be halted. If processes are allowed to fail only once, progress is still guaranteed.

Theorem 6.5 *During a normal execution in which failures (no restarts, not in the critical region, and reinitialized variables) are allowed, if there is some process not in remainder region that never fails, then some process eventually reaches the critical or remainder region 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. ∎

A major deficiency in the Bakery algorithm is the need for unbounded variables. If a process is always in the trying region (i.e. processes cycle through the critical region and back into the trying region, before the trying region has completely emptied), its *number* will grow arbitrarily big. In addition, the use of safe registers will allow reads of *number* to be any possible integer, which can cause assignments of large numbers.

6.2 Peterson-Fischer algorithm

The code for this algorithm is located in Figure 6.3 and Figure 6.5. This algorithm keeps the features of fair progress, mutual exclusion, and single-writer variables that Lamport's algorithm had, but it uses only binary-valued shared variables (as opposed to unbounded variables) and has better failure resiliency (it allows failures to restart). Failed processes simply return to the remainder region with reset initial variables. The fairness condition is, thus, if some process, p, is outside $\mathcal{R}$ and doesn't fail, then eventually p makes a region change. A drawback of this algorithm is that it does require atomic registers with indivisible and non-overlapping read and write operations.

The basic idea of this algorithm is that processes are assigned to leaves of a complete binary tree. The processes then compete at each level to advance up the tree. Once a process

Shared variables:

- q : an array indexed by $\{0,1\}$ of values from $\{nil, \mathrm{T}=1, \mathrm{F}=0\}$, 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 6.3: Peterson-Fisher's 2-process mutual exclusion algorithm

reaches the root, it enters the critical region and is removed from the competition. We begin by first describing the competition between two nodes as they try to climb the tree.

6.2.1 The 2-Process algorithm

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:

	steps of p_0	**steps of p_1**
time	reads $q[1] = nil$	
$\downarrow$		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 any processes i to enter the **wait** loop without having recognized that the other process is also in $\mathcal{C}$ or in $\mathcal{T}$ after the first line of its code (i.e. that $q(opp(i)) \neq nil$). At this point, it is the condition $i \oplus (q[opp(i)] \neq q[i])$ that might allow i to enter $\mathcal{C}$ and not the condition $q[opp(i)] = nil$. Hence, based on the fact that either $q(1) = q(0)$ or not, only one process can take possession of the critical region and mutual exclusion is verified.

Consider for instance how the preceding scenario is modified with the second test:

	steps of p_0	**steps of p_1**
time	reads $q[1] = nil$	
$\downarrow$		reads $q[0] = nil$
	sets $q[0] = \mathrm{T}$	
	reads $q[1] = nil$	
	reads $q[1] = nil$	
	enters $\mathcal{C}$	
		sets $q[1] = \mathrm{T}$
		reads $q[0] = \mathrm{T}$
		sets $q[1] = \mathrm{F}$
		reads $q[1] = \mathrm{F}$
		sees $1 \oplus (\mathrm{T} \neq \mathrm{F})$ is not satisfied
		does not enter $\mathcal{C}$

The 2-process algorithm satisfies fairness according to the following no-lockout condition:

Theorem 6.6 *If some process p is outside $\mathcal{R}$ and doesn't fail, then p eventually enters $\mathcal{C}$.*

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

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

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. ∎

6.2.2 The n-Process algorithm

We now generalize the algorithm to n-processes. The processes compose a tree as shown in Figure 6.4. 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 a process 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 6.5).

Also predetermined is the set of potential opponents for a process i at a given level k, denoted by opponents(i, k) - the set of processes arising from the "brother subtree": the subtree which has the same parent as the node (i, k). 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.

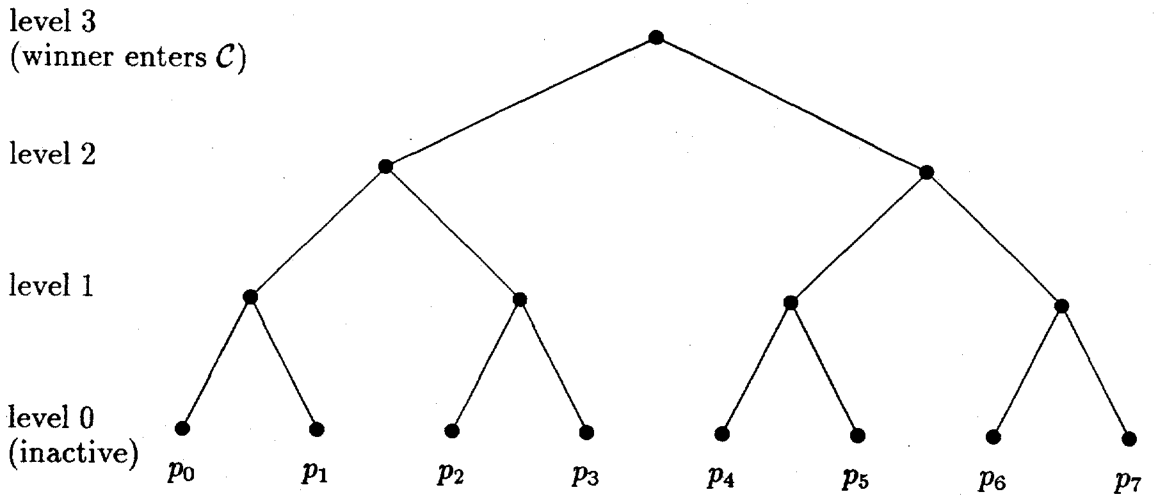


Figure 6.4: The playoff tree for the Peterson-Fischer n-process algorithm on 8 processes.

When a process p_i is ready to compete at level k, it uses the subroutine $OPP(i,k)$ to find its opponent. $OPP(i,k)$ searches through the opponents(i,k) until it finds one then at a level at least as high as k. If the process p has no such opponent or has one whose level is k and whom it beats, it advances to the next round. Otherwise, the **if**-condition is not satisfied or its opponent has already advanced above that level. The process then waits for its **if**-condition to become true.

Mutual exclusion

Theorem 6.7 *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. ∎

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 arbitrary number of times.

Failure resiliency

The Peterson-Fischer algorithm allows for the same type of failures as in the Bakery algorithm (i.e. Failures occur outside the critical region, the process goes back to $\mathcal{R}$, and its registers are reset). However, unlike the Bakery algorithm, process restarts are allowed. Once a process fails, its opponent condition to advance is satisfied. A restarted process can not change this condition.

The next lecture will give time bounds for the Peterson-Fisher algorithm.

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 $j \in$ opponents$(i,k)$ do
  $opp \leftarrow q[j]$
  if level$(opp) \geq k$ then return$(opp)$
return(0,F)
```

Code for p_i

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

$q[i] \leftarrow$ (0,F)

**Remainder region**
```

Figure 6.5: Peterson's n-process mutual exclusion algorithm.

6.852 Distributed Algorithms	**Fall Semester, 1990**

Lecture 7: October 2

Lecturer: Nancy Lynch *Scribe: Rainer Gawlick*[5]

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

In the previous lecture we introduced the Peterson-Fischer tournament algorithm as a generalization of the Peterson-Fischer 2-process algorithm, and proved that it satisfied mutual exclusion. We will now prove that the Peterson-Fischer tournament algorithm satisfies some other desired properties, namely: progress and no lockout. We will then analyze its complexity.

7.1.1 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 (same 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 arbitrary number of times.

7.1.2 Complexity Analysis

We now turn to another important aspect that we have so far overlooked in our discussion of mutual exclusion algorithms, namely *complexity analysis*.

[5]Based on lecture notes from 1988 scribed by Azer Bestavros

7.1.3 Space Analysis

Here we are interested in the number of shared 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[6]. Thus $\mathcal{O}(n\log n)$ values are needed.

7.1.4 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[7].

- 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[8]. Let b be that bound.

- Let $T(k)$ be the maximum time it takes a process[9] 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[10] before it reaches the wait loop, for which two cases can be singled out:

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

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

[8]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.

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

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

1. P_i finds the **if** 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 **if** 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 win over the other one.

   2.1. If p_i is the winner, it need not wait so we get

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

   2.2. 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 at level $k+1$. Thus:

$$\begin{aligned} T(k) &= \mathcal{O}(2^k) + 2T(k+1) + b \\ &= \mathcal{O}(2^k) + 2T(k+1), \end{aligned}$$

   since the constant b is absorbed in the $\mathcal{O}(2^k)$ term.

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

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

- Let us solve the above recurrence. Recall that a is the maximum step time. This constant is absorbed in the $\mathcal{O}(2^k)$ term in the preceding discussion.

$$\begin{aligned} T(0) &\leq a2^0 + 2T(1) \\ &\leq a(2^0 + 2^2) + 2^2T(2) \\ &\leq a(2^0 + 2^2 + 2^4) + 2^3T(3) \\ &\leq \vdots \\ &\leq a(2^0 + 2^2 + \ldots 2^{2(\log n - 1)} + 2^{\log n}T(\log n) \\ &= \mathcal{O}(n^2) + \mathcal{O}(n) = \mathcal{O}(n^2). \end{aligned}$$

Hence we get the bound $T(0) \leq \mathcal{O}(n^2)$ on the time from when a process enters $\mathcal{T}$ to when it enters $\mathcal{C}$.

7.2 Mutual Exclusion Requires n read/write Variables

The following impossibility result, due to Burns and Lynch, provides a lower bound on the space complexity of the mutual exclusion problem in the atomic register multi-read/write model. Specifically, Burns and Lynch show that *any* mutual exclusion algorithm in this model must use at least n shared variables:

Theorem 7.1 (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.*

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.

Let us first consider the case were the variables are single writer variables. The result is then obvious because every process needs to be able to write something in order to convey information to other processes.

But how can we proceed with general read write systems where each variable could be written and read by any process?

The basic idea for the multi read/write case is to assume that there is an algorithm that uses fewer than n shared variables 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 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.

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

Without loss of generality we assume that all processes are *deterministic* i.e.: there is only one locally controlled action enabled at each state, and only one possible next state for a given state and action.

If we have a state q and a sequence of process indexes h, called a schedule, there exists a unique state q' which results from running one step for a process whenever that process' index occurs in the schedule.

We define $q' = result(q, h)$.

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. Process p_i is *hidden* from q by h if the original state is q and if h can be written as $h = h_1 h_2$ such that 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 makes this fact formal:

Lemma 7.2 *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 7.1 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:

Lemma 7.3 *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 $q = result(q_0, h)$.*

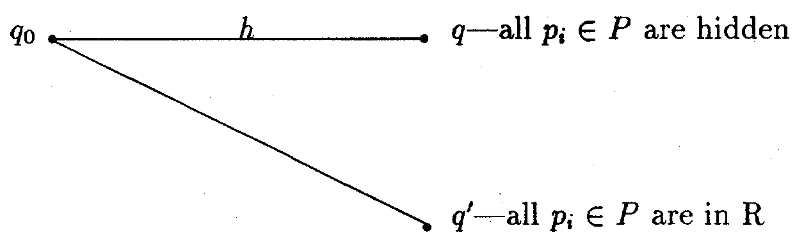
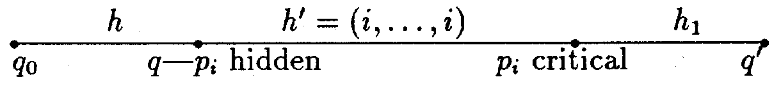


Figure 7.1: Hidden processes do not affect other processes. Here, if $p_i \in P$ are hidden during h, then q and q' agree in all respects except for the local state of the processes in P.

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 every 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 7.2: 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. ■

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 8: October 4

Lecturer: Nancy A. Lynch *Scribe: Philippe Park*[11]

8.1 Burns-Lynch's read/write Lower Bound (Cont.)

Lecture 7 covered most of the initial lemmas needed to prove that n variables are needed to provide mutual with progress exclusion for n processes in the read/write atomic register model.

The basic strategy of the proof is to show that a process p in the remainder region must write to some variable which is not *covered* before going into the critical region. If it did not write such a variable, then we could construct an execution where it would not communicate to the other processes that it was entering into the critical region; expressed in another way, the rest of the system could not distinguish a configuration with p in R from one with p in C. A fortiori, the same result would occur if the variable were *obliterated* before p entered C. The terms *hidden*, *covered*, and *obliterated* were defined in the last lecture to make the notion of "communicating information" more explicit.

In this lecture, we will finish the proof. The goal will be to show that if processes must write some variable not covered by any other process to satisfy mutual exclusion, then there must be n separate variables covered by n processes.

We begin by introducing a new term, *nullified*, which is defined below. We combine the definitions of *covered* and *hidden*. 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 finite history, and $q = result(q_0, h)$. A variable v is *nullified* by process p_i in the execution from q_0 by h if

1. p_i covers v in state q; and

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

•

Now we can show that, in any algorithm solving mutual exclusion, there is a finite history yielding n distinct nullified variables. The Burns-Lynch theorem of a lower bound on the number of shared variables will then follow immediately.

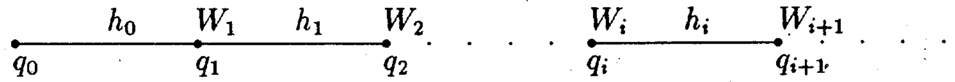
[11]Based on lecture notes from 1988 scribed by Jon Riecke

Lemma 8.1 *Let S be a system with more than 2 processes solving mutual exclusion with progress. Let q_0 be any reachable configuration in which all processes are in R. Then for every k, $1 \leq k \leq n$, there is a finite history h using only processes $p_1, \ldots, p_k$, such that k distinct variables are nullified by $p_1, \ldots, p_k$ in the execution from q_0 by h.*

Proof: Proceed by induction on k. In the base case, $k = 1$. Since S verifies progress, running p_1 alone means that p_1 eventually reaches C; call this finite history h'. By the covered-variable-lemma, p_1 must write some non-covered variable during this execution. Stop h' just before p_1 writes *some* variable v (covered or non-covered) and call this finite history h. In this finite history 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 execution from q_0 by h.

In the induction case, assume the lemma holds for $k-1$. Then there is a finite history 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 $q_1 = result(q_0, h_0)$.

Repeating the induction, we can construct the sequence shown below, where for each h_i, there is at least one step of $p_1, \ldots, p_{k-1}$, and for each i, the $k-1$ variables in W_i are nullified by $p_1, \ldots, p_{k-1}$ from q_{i-1} by h_{i-1}.



Here is how we make the induction step work. Assume that q_{i-1} is given. Assume an execution h_{i-1} beginning with indices $1, 2, 3, \ldots, k-1$, and then let $p_1, \ldots, p_{k-1}$ take steps until they reach the remainder region. This is possible due to the progress condition. Then use the inductive hypothesis again to run $p_1, \ldots, p_{k-1}$ alone until they nullify a set of distinct variables , W_i, in the execution from q_{i-1} by h_{i-1}. We can argue that this is reasonable by using an intermediate step in the following way: in the state q'_{i-1}, all of the processes $p_1, \ldots, p_{k-1}$ are in the remainder region and are hidden from q'_{i-1}. This implies that they are also hidden from q_{i-1}.
Somehow, p_k must get involved in this execution so that it nullifies a distinct variable. We will do this by contradiction. Let us start by constructing some "side branches". (See Figure 8.1) The side branches are finite histories s_i which proceed from state q_i and involve only p_k. By the hidden-lemma, 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, are the same in q' as in q_i.

In other words q' "looks like" q_i from the perspective of p_k. From q', p_k must be able to reach C on its own, since the algorithm verifies progress and all processes are in R. Thus,

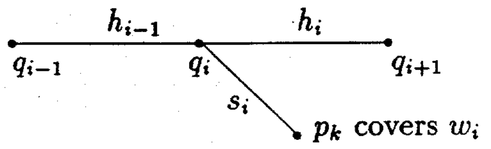


Figure 8.1: The construction of side branches. Note that s_i only involves process p_k.

p_k can also reach C on its own from q_i, since q_i "looks like" q'. Call such a finite history 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 during h_{i-1}. By the covered-variable-lemma, p_k must write some variable v during the execution 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 prefix of s_i' where p_k covers some variable $w_i \notin W_i$.

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 indices i and j with $w_i = w_j$; we know that these indices exist by a pigeonhole argument, since we can construct a very long chain (length $> k$) of the q_i's (see Figure 8.2).

Create a finite history h, such that

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

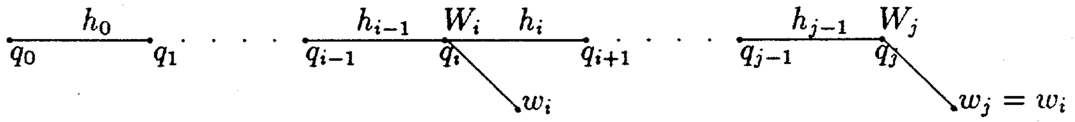


Figure 8.2: The construction of the finite history h. Process p_k covers $w_i = w_j$ in the two side branches depicted.

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. ■

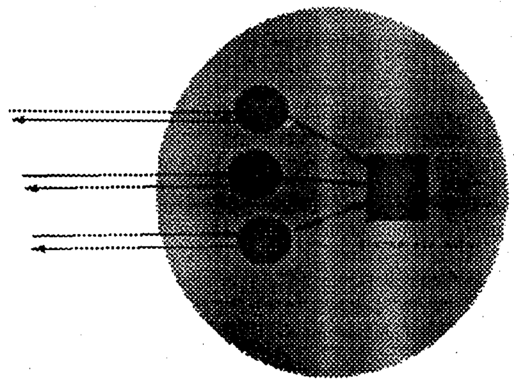


Figure 8.3: Shared Memory as one Big I/O Automaton

8.2 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.* This is a more powerful model where *read-modify-write* operations are all done atomically. This model assumes a broad range of computation which is much more extensive than the single word comparison (i.e. the "test") which is normally available in today's computers. Since this model would need some low-level arbitration for a real hardware implementation, providing mutual exclusion for structures beyond simple shared memory cells might be difficult.

Now, if we consider mutual exclusion assuming the test-and-set primitive, we note that a small change in the model makes a drastic difference in the kinds of impossibility results one can get. It would seem that mutual exclusion would be trivial to show; that intuitively fair exclusive access to shared memory implies fair exclusive access to the critical region. But it turns out that this assumption does not make things trivial as for instance our next lower bounds result will demonstrate.

Let's begin with a brief description of our indivisible test-and-set model for accessing shared memory. Assume that all shared memory collapses into a single shared variable. Although this would be very difficult to model with today's hardware, it simplifies the model and makes it very powerful.

In the automaton described by figure 8.3 indivisible process steps involve arbitrary changes to the state of p_i, and the shared variable, based on their prior states. This whole system can be modeled as one big I/O automaton as in the Burns-Lynch model with one process per class.

In order to see how this model is different from the read and write model, consider figure 8.4.

The wait operator shown in 8.4 is a "busy" wait, continuing to check the variable v; while the condition is true, the process executing this code doesn't do anything. However,

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

Code for p_i:

```
waitfor $v = 0$
$v \leftarrow 1$
**Critical region**
$v \leftarrow 0$
**Remainder region**
```

Figure 8.4: A trivial test-and-set algorithm for achieving mutual exclusion

in describing this operation, one has to be careful about the indivisibility of actions. The variable v is repeatedly tested for value of zero. During the actual "test" operation, no other access by other processes is permitted. After each iteration where the condition is found to be false, the variable is released. As soon as the condition is found to be true (i.e. v=0), then *without releasing variable* v, v is set to 1, and the variable is again released after the "set" operation.

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

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

The test of the while loop is done on the *locked* variable. Figure 8.5 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.

It is important to note that locks can only be placed around statements in the trying or exit regions, not the critical or remainder regions, since these are the program sections where the transitions in and out of critical sections occur. In the critical section, locking a shared variable would be forbid in an unnecessary way the other processes to take steps; in the remainder region, a resource would be tied up when it was not being used.

Notice how a small change in the underlying model makes a drastic changes in the

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 8.5: A trivial test-and-set algorithm (rewritten)

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.

8.2.1 Burns-Lynch's test-and-set Lower Bound (Cont.)

In the simple algorithm given above, lockout is possible. However, we can get fairness back by forcing FIFO behavior. This can be done by replacing the variable v with a queue. Now, at the beginning of T, a process adds itself to the queue (atomically), and waits for its turn to enter C. On the other hand, when leaving C, a process removes itself from the queue, thus enabling the next process to enter 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. For another way to see this bound, say that a queue can have $n!$ values. But Stirling's formula expresses that for any ϵ, $(n-\epsilon)^n \leq n! \leq n^n$. then

One way of reducing the space requirement in the above solution is to have each process grab a ticket as it enters T and then waits for its turn to enter C. This solution requires a shared variable with two values:

1. *next_ticket*, the next ticket to be issued

2. *permitted_ticket*, the ticket permitted to enter C

At the beginning of 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 C, it will have to increment the *permitted_ticket* field of the shared variable. All executions are done modulo 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 achieve mutual-exclusion, progress, and bounded waiting[12].

Theorem 8.2 *Let $\mathcal{P}$ be a protocol involving at least n processes, $n \geq 1$, and q be any configuration. Assume $\mathcal{P}$ satisfies mutual exclusion, progress, and bounded waiting. Then, the shared variable of $\mathcal{P}$ takes on at least n values.*

Proof:

- Let $V(q)$ be the values of the shared variable in state 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 finite history h, we define $result(q, h)$ to be the configuration that results from applying the finite history h starting from configuration q.
- Let $exit(q)$ be a finite history that would result in having all processes in R, where only the processes not in R in configuration q are allowed to appear in the finite history. Note that by the normal operation assumption and the fact that $\mathcal{S}$ guarantees progress, this finite history is always possible.
- We define $enter(q, i)$ to be the finite history that contains steps taken only by process p_i, starting from a configuration q in which all processes are in R. Thus, p_i will eventually end up in C.
- Let $q_0 = result(q, exit(q))$. In q_0 all processes are in R.
- Let $q_1 = result(q_0, enter(q_0, 1))$. In q_1, process p_1 is in C, whereas all other processes are still in R.

[12]no claims about how many times a process in T will be bypassed.

- Now, let each process p_j, $2 \le j \le n$, enter T in turn. This can be done by the following sequence of configurations: $q_j = result(q_{j-1}, j)$, where $2 \le j \le n$.
- We claim that $V(q_i) \ne V(q_j)$, for $0 < i < j \le n$. We prove this claim by contradiction as follows:
  - Assume that for some i and j, where $0 < i < j \le n$, 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 finite history h that yields a normal execution and which involves only $p_1, p_2, \ldots, p_i$ and results in some process entering C infinitely many times.
  - The same finite history 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 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 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 $V(q_i) \ne V(q_j)$, for $0 < i < j \le n$. Thus, we need at least n values for the shared variables used in $\mathcal{S}$.

■

8.2.2 An upper bound about test-and-set Algorithms

In the last section, we determined a lower bound of n on the number of values the shared test-and-set variable had to take for any algorithm satisfying mutual exclusion and bounded waiting. Could this lower bound be raised to $\Omega(n^2)$? Another result of Burns-Fischer-Jackson-Lynch-Peterson shows that this is not the case: they construct an algorithm that achives mutual exclusion and does not require this many test-and-set variables. This algorithm is not of much interest per se because it is not very practical. The main interest is in showing a counterexample to an impossibility conjecture. On the other hand this algorithm uses only one shared variable that stores only $n + 6$ different values. This result, along with the lower bound of n proven in the previous section shows that this algorithm is quite close to optimal in terms of storage requirements among the algorithms using test and set variables. The basic idea is illustrated in Figure 8.6.

Processes entering region T go into a *"Buffer"*, where they lose their relative arrival order. They accumulate here until the *"Main"* section becomes empty, whereupon they all proceed to *Main* at once. Here, they proceed into the critical region one at a time, in an ordered fashion. Any process will be bypassed at most once by any other process, so that

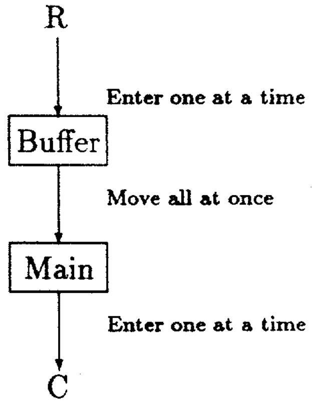


Figure 8.6: A conceptual view of the Burns, *et al.* algorithm for mutual exclusion.

this algorithm meets the bounded waiting condition. Mutual exclusion is ensured by the fact that a BYE message is in the shared variable only if the critical region is free, that an ELECT message is sent only when a BYE message has been detected, and that at most one process at a time has access to the shared variable due to the test and set feature.

Some communication mechanism telling processes when to change regions is needed to make this work.

One solution is to use the abstraction of a *supervisor* process. We can think of this process as being around at all times. If *Main* is empty it then moves all of the processes from *Buffer* to *Main*. Then it dispatches the processes in *Main* to the critical region, one at a time.

Actually, 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.

Now let's take a look at the strategy in more detail.

First, let the shared variable have two fields. One field has for range of values $0, \dots, n$ and represents the *count* of the number of processes that have entered the trying region. The other field holds one of the messages ENTER, ELECT, BYE and ACK. This implies an upper bound of $c \cdot n$ $(c = 4)$ on the number of values taken by the shared variable.

The algorithm proceeds as follows. When a process enters T, it increments the first field *count* of the variable to inform the supervisor that a new process has entered T, then waits in *Buffer*. When the supervisor gains control of the shared variable, it can see how many processes have entered T, since the last check. The supervisor then adds this count to its local variable *buff*, and resets *count* in v to be 0. If *Main* is empty, the supervisor moves all of the processes from *Buffer* to *Main*; if there are processes in *Main*, it moves each process in

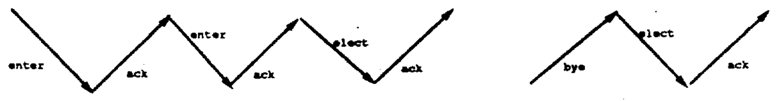


Figure 8.7: An example of a thread of messages

turn into the critical region and waits until that process announces that it has left the critical region. All these communications are done by "sending messages" via the shared variable. The supervisor may send the messages ENTER and ELECT to the processes: ENTER tells a process to move from the "buffer" to the "main" sections, and ELECT tells a process to proceed from the "main" section to its critical region. Any process waiting for a specific message from the supervisor may pick it up as soon as it sees it in the shared variable. The processes themselves send ACK messages acknowledging the receipt of the ENTER and ELECT messages, and BYE messages when they leave their critical regions.

The messages are appearing sequentially in the second field of the shared variable: each ENTER or ELECT is followed by a corresponding ACK, and there is always a BYE in between two ELECT. See figure 8.7 for an example of such a thread of messages.

Actually, the algorithm is optimized in order to reduce the size of the value set of the shared variable from $c.n$ to $c + n$ and get the result we promised earlier. c is the number of special messages. In the final version of the algorithm c will be 6. In this first optimized version the variable v has only *one* field and is used to communicate the thread of special messages (at most c different values) *or* the number of processes newly arrived in T (at most n different values). We implement this in the following way. If a process p_i needs to update a count whereas the variable v contains a special message ENTER, ELECT, ACK or BYE, p_i will "steal" the message, "breaking" for a while the thread, write a 1 to v, and wait for v to be reset to 0. Each process p_i will maintain a variable in which it will keep a local copy of any message that it steals from the shared variable. (We can check that any process cannot have stolen two messages at a given point of the execution without having already returned the latter one.) Then, if p_i ever reads v and sees a 0, it can return the stolen message to v so that the thread of messages can resume. To see that p_i must indeed 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 stolen message to v. A similar quiescent state will occur in the other two cases. Figure 8.8 gives code for this simplified description of the algorithm.

The algorithm requires yet one more refinement to distribute equally the role of "supervisor" among all processors and get a truly distributed algorithm. 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. Hence in the exit protocol, the current supervisor (call it supervisor 1) passes the job of "supervisor" on to another process (call it supervisor 2) and sends it all the necessary state information. This state information must be passed through the shared variable as well. The message COUNT is used by supervisor 1 to let know supervisor 2 of the number of other processes that are queueing in Main. There is one special case to consider if no one is around the king (i.e., *buff* = *main*= $v = 0$) when time has come for him to retire from the Critical Section. No prince can then be selected and v is set to FREE. The complete algorithm is given in Figures 8.9 and 8.10.

8.2.3 Another Lower Bound about test-and-set Algorithms

Lower Bounds for No Lockoutlower bounds 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 $\frac{n}{2}$. There is a somewhat technical restriction in this theorem that does not quite cover all no-lockout test-and-set algorithms: the algorithms considered forbid processes entering in the Trying region to remember anything from prior executions. (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.

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 $v \in \mathcal{N}$ then $v \leftarrow v + 1$
   else [$m \leftarrow v$; $v \leftarrow 1$; waitfor $v = 0$; $v \leftarrow m$;]
waitfor $v$ = ENTER; $v \leftarrow$ ACK;          ** Buffer **
waitfor $v$ = ELECT; $v \leftarrow$ ACK;          ** Main **
unlock;

  ** Critical Section **

waitfor $v = 0$; $v \leftarrow$ 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
         $buff \leftarrow buff + v - 1$; $v \leftarrow$ ENTER; $main \leftarrow main + 1$;
         while $v \neq$ ACK do
            if $v \in \mathcal{N}$ then
               $buff \leftarrow buff + v$; $v \leftarrow 0$;
            unlock; lock;
         $v \leftarrow 0$;

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

while $main > 0$ do
   $main \leftarrow main - 1$; $v \leftarrow$ ELECT;
   while $v \neq$ ACK do
      if $v \in \mathcal{N}$ then
         $buff \leftarrow buff + v$; $v \leftarrow 0$;
      unlock; lock;
   while $v \neq$ BYE do
   if $v \in \mathcal{N}$ then [$buff \leftarrow buff + v$; $v \leftarrow 0$;]
   unlock; lock;
goto L;
```

Figure 8.8: 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 $v \leftarrow 0$
      else
        if $v \in \mathcal{N}$ then $v \leftarrow v + 1$
          else [$m \leftarrow v$; $v \leftarrow 1$; waitfor $v = 0$; $v \leftarrow m$; $m \leftarrow 0$;]
        waitfor $v$ = ENTER; $v \leftarrow$ ACK;          ** Buffer **
        waitfor $v$ = ELECT; $v \leftarrow$ ACK;          ** Main **
        while $v \neq$ BYE do          ** Receive Counts **
          if $v \in \mathcal{N}$ then [$buff \leftarrow buff + v$; $v \leftarrow 0$;]
          if $v$ = COUNT then [$main \leftarrow main + 1$; $v \leftarrow$ ACK;]
          unlock; lock;
        $v \leftarrow 0$
    unlock;

     ** Critical Section **

    lock;
```

Figure 8.9: Burns, *et al.* Mutual Exclusion Algorithm, trying protocol.

Exit Protocol:

```
if $main = buff = v = 0$ then $v \leftarrow$ FREE
  else
    if $main = 0$ then        ** Move processes from buffer to main: **
      while $buff + v > 0$ do
        $buff \leftarrow buff + v - 1$; $v \leftarrow$ ENTER; $main \leftarrow main + 1$;
        while $v \neq$ ACK do
          if $v \in \mathcal{N}$ then
            $buff \leftarrow buff + v$; $v \leftarrow 0$;
          unlock; lock;
        $v \leftarrow 0$;

 ** Elect new supervisor: **

    $buff \leftarrow buff + v$; $v \leftarrow$ ELECT; $main \leftarrow main - 1$;
    while $v \neq$ ACK do
      if $v \in \mathcal{N}$ then
        $buff \leftarrow buff + v$; $v \leftarrow 0$;
      unlock; lock;

 ** Send counts: **

    while $main > 0$ do
      $v \leftarrow$ COUNT; $main \leftarrow main - 1$; waitfor $v =$ ACK;
    $v \leftarrow buff$;
    waitfor $v = 0$; $v \leftarrow$ BYE;
unlock;

 ** Remainder region **

lock;
```

Figure 8.10: Burns, *et al.* Mutual Exclusion Algorithm, exit protocol

6.852 Distributed Algorithms	**Fall Semester, 1990**
Lecture 9: October 12	
Lecturer: Isaac Saias	*Scribe: Roberto Segala*[13]

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 process 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 process 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 process 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 processes 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].

Many things have happened since the lecture has been given. In his original proof Rabin had neither provided a formal statement of the correctness statement nor a proof for it. In this course we actually attempted to fill in the gaps and provide a mathematically complete analysis of the algorithm. By the time of the lecture we had already discovered a problem with the strong correctness version proposed by Rabin. (His statement corresponds to our Theorem of page 95 where c/n is replaced with c/m.) Since then we discovered another problem which invalidates the weaker result c/n that we give. (We have been able to prove that the adversary is actually so strong that it can lockout up to a linear fraction of the number of the processes with a probability exponentially close to one.) The problem involved is of a very subtle nature and is due in essence to the fact that the probability measure involved in the no-lockout statement is partially controlled by the adversary.

We present nevertheless the notes of the class as it was taught: the techniques involved are still very useful and by the time this was written even Rabin was unaware of the problem!

A complete and (correct) analysis is presented in the MIT Technical Report:
MIT/LCS/TM-462.

[13]Based on lecture notes from 1988 scribed by Mark Tuttle

9.1 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 process takes its steps only when certain shared variables are set to certain unlucky values that keep the process 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 process 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 processes 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 processes 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 process 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 process steps occurring during the execution, and the sequence of coins flipped by the processes 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 process steps? We often think of the choice of the next process 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 processes take steps. In general, we let the adversary control those factors (such as the processes' 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 processes 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 process 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 processes 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 process 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 process 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.

9.2 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 $(\log n)$ 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 process draws a number at random, and the process 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 process 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 process 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 $(\log n)$ bound on the number of values assumed by V, Rabin takes b to be $\lceil \log n \rceil + 5$ and r to be a small constant such as 99.

The algorithm itself appears in Figure 9.1. To enter the critical region, process p_i chooses a lottery number b_i from $\{1, 2, \ldots, b\}$ at random according to the probability distribution

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

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 K flips (that is, if the coin comes up tails K flips in a row). Having chosen its lottery number B_i, p_i updates the maximum lottery number chosen by a process during the current round, a value posted in B, by setting B to the maximum of B_i and B. The process 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 processes competing for the chance to 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 processes 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 processes 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 [14]. Process p_i records the

[14]Note that we have modified the code in a minor way to make the analysis uniform: in our version of the algorithm the round number is initialized to *random*.

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

```
    $S \in \{0,1\}$, initially 0                                    ** semaphore **
    $B \in \{0,1,\ldots,\lceil \log n \rceil + 5\}$, initially 0    ** posted number **
    $R \in \{0,1,\ldots,99\}$, initially *random*                   ** 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 $V \neq (0, B_i, R_i)$ do        ** if not winner at same time C is available **
   if $(V.R \neq R_i)$ or $(V.B < B_i)$ then      ** not yet participated in lottery **
      $B_i \leftarrow random$
      $V.B \leftarrow max(V.B, B_i)$
      $R_i \leftarrow V.R$
   unlock; lock;
$V \leftarrow (1, 0, random)$
unlock;

  ** Critical Region **

lock;
$V.S \leftarrow 0$
$R_i \leftarrow \perp$
$B_i \leftarrow 0$
unlock;

  ** Remainder Region **

lock;
```

Figure 9.1: Rabin's randomized mutual exclusion algorithm.

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 process to enter the critical region has done so). Then since B and R are reset only when a process 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.

Note that a process i might enter the Trying region with its round number R_i being equal to $V.R$, and with its lottery number B_i being equal to $V.B$ at its time of entry in T. In some sense, the new comer i then "steals" the success from a previous competitor waiting to be called again by the scheduler.

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 guarantees progress since some process wins every round and is able to proceed to the critical region. Neither of these statements is a probabilistic statement. The algorithm also satisfies a probabilistic version of no lockout saying roughly that if a process 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 process allowed to take a step as a deterministic function of the sequence of past processes' steps and region 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 process p_{i_j} and (old_j, new_j) denotes a couple of regions (e.g. $(\mathrm{Try}_j, \mathrm{Try}_j)$ or $(\mathrm{Try}_j, \mathrm{Crit}_j)$). Such a run is said to be the run of an execution if i_j denotes the process p_{i_j} that takes the jth step in the execution and (old_j, new_j) denotes the region change $old_j \rightarrow new_j$ process p_{i_j} undergoes 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 process 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 = Remainder (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 process 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 process playing the lottery is equally likely to win, and hence each process 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 process enters its critical region until the time the next process enters its critical region; formally, a *round* of an execution is a maximal execution fragment for the given execution containing one transition Try $\rightarrow$ Critical at the end of the execution fragment and containing no other transition Try $\rightarrow$ Critical. We say that a process p_i *participates* in a round if a transition Try $\rightarrow$ Critical or Remaiı der $\rightarrow$ Try or Try $\rightarrow$ Try by p_i appears in that round. We say that a process p_i *participates* in a round if a transition Try $\rightarrow$ Critical or Remainder $\rightarrow$ Try or Try $\rightarrow$ Try by p_i appears in that round.

The statement of the no lockout condition is as follows:

Theorem 9.1 *Let $k \geq 1$. For every normal adversary A and every $(k-1)$-round run α compatible with A, the probability that p_i enters the critical region in round k, given that p_i participates in round k of an execution compatible with α, is at least c/n for some constant c.*

For simplicity, from now on we will refer to process p_i as i.

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

Fix $b = \lceil \log n \rceil + 5$. Let A be a normal adversary, $(\rho(k))_{k=1}^{\infty}$ an infinite sequence of numbers in $\{0, ..., 99\}$ and let $(\beta_i(k))_{k=1}^{\infty}$, $i = 1, \ldots, n$ be n infinite sequences of numbers in $\{1, ..., b\}$. Then we define $Exec_{\mathcal{A}}(\rho, \beta_1, \ldots, \beta_n)$ to be the execution of the algorithm with adversary A, in which the successive choices of round numbers are drawn from ρ and the successive choices of the B_i's are drawn from β_i. More specifically, the inspection of the algorithm shows that each processor i participating in a round k draws a new lottery number at most once during that round. If it does, we can think of it using $\beta_i(k)$ for that purpose. In the same way, the algorithm uses $\rho(k)$ as the round number for round k; for $k = 1$ this number is chosen in the initialization, while for $k \geq 2$ it is chosen at the very end of round $(k-1)$ by the processor that wins round $k-1$.

For any given execution ω we will consider various quantities defined in terms of the values of the local and shared variables of the algorithm. Let X be such a variable. We will denote by $X(k,\omega)$ the value in variable X just prior to the last step (Try $\rightarrow$ Critical) of round k of execution ω. When no confusion is likely, we will omit the parameter ω, writing, for example, $R(k)$ in place of $R(k,\omega)$. Special cases of X that we will use are R, R_i, B and B_i, where the subscript i ranges over $1,\ldots,n$.

Let's reiterate the meanings of all these variables.

- $R(k)$ is the round number used during round k.
- $R_i(k)$ is the round number process i ends up with at the end of round k.
- $B_i(k)$ represents the lottery number process i ends up with at the end of round k; it is conceivable that process i does not set $B_i(k)$ to $\beta_i(k)$ (so that $B_i(k) = B_i(k-1)$), *even* if it participates in round k. This explains why we have chosen a different notation $\beta_i(k)$ to deal with the actual lottery numbers.
- $B(k)$ represents the lottery value of the process that ends up winning round k. Recall that this winning process might *not* be the one that last updated B by doing $V.B \leftarrow max(V.B, B_i)$.

The convention of denoting $X(k)$ the value of variable X by the end of round k can be extended to variables that are not defined only in terms of the value of the local program variables at this time. For instance $N(k)$ will represent the number of processes participating in round k. Note that this quantity is not completely in the control of the scheduler if this scheduler give steps to processors not already in the trying region while the semaphore $V.S$ is equal to 0. For instance the scheduler cannot ensure with certainty that more then 2 processes enter in the Trying region at this point: the first process doing so could go right through into the critical region!

Let's introduce and define also some events that will appear in the course of the analysis.

- For each i and k we can define $New_i(k) = \{p_i$ chooses a new value during round $k\}$
- For each k define $Allnew(k) = \{$All the processes j participating in round k chose a "new" value $B_j(k)$ during round $k\}$. $Allnew(k)$ is the event where all participating processes j verify that $(V.R \neq R_j(k-1))$ or $(V.B < B_j(k-1))$ at their first step within round k.
- Define also $U(k)$ (U stands for Uniquemax) to be the event: $\{$The number of j's participating in round k such that $B_j(k) = B(k)$ is one$\}$. $U(k)$ is the event where at most one process ends up with the highest lottery value in round k.

Call $i_1(k,\omega) = i, i_2(k,\omega), \ldots, i_{N(k,\omega)}(k,\omega)$ the indices of the processors participating in round k of execution ω. Then the preceding events are related by

$$Allnew(k) = \bigcap_{l=1}^{N(k)} New_{i_l}(k)$$

Let us now turn to the probabilistic aspect of the model, introduce the random inputs of the algorithm and discuss the probability induced on the space $Exec_{\mathcal{A}}(\rho, \beta_1, \ldots, \beta_n)$.

The sequence ρ is obtained as a sequence of iid (independent identically distributed) uniform random variables, the sequences $(\beta_i(k))_{k=1}^{\infty}$ are constructed as sequences of iid truncated exponential random variables:

$$\begin{aligned} \Pr\Big[\rho(k) = l\Big] &= \tfrac{1}{100} \quad 0 \leq l \leq 99, \\ \Pr\Big[\beta_i(k) = l\Big] &= \begin{cases} \frac{1}{2^l} & \text{if } 1 \leq l < b \\ \frac{1}{2^{l-1}} & \text{if } l = b, \end{cases} \end{aligned}$$

For any given adversary, the product measures controlling ρ and the β_i's then endow the set of executions with a probability measure. The quantities $N(k)$, $R(k)$, $B(k)$, $R_i(k) \ldots$ can then be viewed as random variables, and the events $U(k)$, $Allnew(k)$, $New_i(k)$, previously defined can be viewed as random events.

The $R(k), k = 1 \ldots$ are iid, since each $R(k)$ is chosen to equal $\rho(k)$ and the $\rho(k)$'s are iid.

The random variables $B_i(k)$ do not have the same distribution as the $\beta_i(k)$. In particular the $B_i(k)$'s are not iid whereas the $\beta_i(k)$ are.

Note that the variables $(B_i(k))_k^{\infty}$ and $(R_j(k))_k^{\infty}$ for arbitrary i and j's are not independent.

The analysis of the algorithm will also make use of the iid random variables ρ_i', $i = 1 \ldots n$, taking value in the set of integers and whose distribution is given by

$$\Pr\Big[\beta_i'(k) = l\Big] = \frac{1}{2^l} \quad l = 1, 2 \ldots$$

The variables $\beta_i'(k)$ can be realized as the number of consecutive (fair) coin flips that one has to wait for in order to get a head. The random variables β_i have the same law as $\text{Min}(\beta_i', b) = \beta_i' \wedge b$.

Even though the underlying probability space we begin working on is the space of executions, we will be able by successive conditionings to reduce the analysis to the *independent* quantities β_i, β_i' and ρ.

Let's restate Rabin's theorem in terms of our notations.

Theorem *There exists a universal (i.e. independent of k and n) constant c such that for every normal adversary A, every $(k-1)$-round run α compatible with A, and every i and m,*

$$\Pr\Big[W_i(k) \mid B(\alpha, p_i, k)\Big] \geq c/n.$$

Note that the previous theorem provides a lower bound of c/n instead of the stronger one c/m claimed by Rabin.

Proof:

In the rest of this writeup k, i, m and α will be fixed quantities and the probability space that we will consider will be the space of executions $Exec_{\mathcal{A}}$ under the probability $\Pr\Big[\ \mid B(\alpha, p_i, k)\Big]$.

In round k, the winning process chooses a new lottery value or keeps an old one (this is a tautology!). By restricting our observation to the event $\mathcal{N}ew_i(k)$ that we previously introduced we can write

$$\begin{aligned}\Pr\Big[W_i(k) \mid B(\alpha, p_i, k)\Big] &\geq \Pr\Big[W_i(k), \mathcal{N}ew_i(k) \mid B(\alpha, p_i, k)\Big] \\ &= \Pr\Big[\mathcal{N}ew_i(k) \mid B(\alpha, p_i, k)\Big] \\ &\qquad \Pr\Big[W_i(k) \mid \mathcal{N}ew_i(k)\ ,\ B(\alpha, p_i, k)\Big].\end{aligned}$$

From the code we have that a process i participating in round k chooses a new value if $R(k) \neq R_i(k-1)$ or $V.B < B_i(k-1)$. We consider here $V.B$ at the point of the execution within round k at which p_i makes its first test on the locked shared variable V. (Note that the set of processes participating in round k is exactly the set of processes that go at least once through the **while** loop of their code during round k. Recall also that $V.B$ grows within round k from the value 0 it takes at the beginning of the round to the value $B(k)$ it assumes at the end.) These considerations allow us to write

$$\begin{aligned}&\Pr\Big[\mathcal{N}ew_i(k) \mid B(\alpha, p_i, k)\Big] = \\ &\qquad \Pr\Big[R(k) \neq R_i(k-1) \cup B(t) < B_i(k-1) \mid B(\alpha, p_i, k)\Big] \\ &\quad \geq \Pr\Big[R(k) \neq R_i(k-1) \mid B(\alpha, p_i, k)\Big] = .99,\end{aligned}$$

where the last equality is formally established in the following claim

Claim 9.2

$$\Pr\Big[R(k) \neq R_i(k-1) \mid B(\alpha, p_i, k)\Big] = .99.$$

Proof:

$$\begin{aligned}
&\Pr\Big[R(k) \neq R_i(k-1) \mid B(\alpha, p_i, k)\Big] \\
&= \sum_r \Pr\Big[R(k) \neq R_i(k-1) \mid R_i(k-1) = r\Big] \\
&\qquad \Pr\Big[R_i(k-1) = r \mid B(\alpha, p_i, k)\Big] \\
&= .99 \sum_r \Pr\Big[R_i(k-1) = r \mid B(\alpha, p_i, k)\Big] = .99,
\end{aligned}$$

where the first equality comes from the fact that the random variable $R(k)$ is independent of all the past i.e. of the $\rho(t)$ and $\beta_j(t); t = 1, \ldots, k-1, j = 1, \ldots, n$ and hence independent of $B(\alpha, p_i, k)$; the second equality comes from the fact that the law of $R(k)$ is uniform. ∎

Thus,

$$\Pr\Big[W_i(k) \mid B(\alpha, p_i, k)\Big] \geq .99 \; \Pr\Big[W_i(k) \mid New_i(k), B(\alpha, p_i, k)\Big].$$

The following result is established in the Appendix.

Claim 9.3

$\forall j = 1, \ldots, n \quad \forall l = 1, \ldots, b$

$$\Big[\Pr\Big[B_j(k) \geq l \mid B(\alpha, p_i, k)\Big] \leq \Pr\Big[\beta_j \geq l\Big]\Big]$$

$$=$$

$$\Big[\Pr\Big[B_j(k) \geq l \mid \neg New_j(k), B(\alpha, p_i, k)\Big] \leq \Pr\Big[\beta_j \geq l\Big]\Big]$$

Lemma 9.4

$$\Pr\Big[W_i(k) \mid New_i(k), B(\alpha, p_i, k)\Big] \geq \Pr\Big[W_i(k) | \text{Allnew}(k), B(\alpha, p_i, k)\Big]$$

Proof:

Among the m processes that participate in round k, some will have chosen a new lottery number and the others (which have necessarily last played and lost in a round with the same round number R), will have kept their old lottery value. Since these processes have lost previous rounds, intuitively their numbers must tend to be rather small. That is,

$$\Pr\Big[B_j(k) \geq l \mid \neg New_i(k), B(\alpha, p_i, k)\Big] \leq \Pr\Big[\beta_j(k) \geq l\Big]$$

or equivalently, as we saw in the previous claim,

$$\Pr\Big[B_j(k) \geq l \mid B(\alpha, p_i, k)\Big] \leq \Pr\Big[\beta_j \geq l\Big].$$

Hence these processes constitute less of a challenge for i. conditioning on $Allnew(k)$ ensures that all the other $m-1$ processes participating to round k constitute a "real" challenge to i by choosing a new lottery number and not keeping an old lottery number. ■

This lemma allows us to write

$$\begin{aligned}\Pr\Big[W_i(k) \mid New_i(k), B(\alpha, p_i, k)\Big] &\geq \Pr\Big[W_i(k) | Allnew(k), B(\alpha, p_i, k)\Big] \\ &\geq \Pr\Big[W_i(k), U(k) \mid Allnew(k), B(\alpha, p_i, k)\Big] \\ &= \Pr\Big[W_i(k) \mid U(k), Allnew(k), B(\alpha, p_i, k)\Big] \\ &\quad \Pr\Big[U(k) \mid Allnew(k), B(\alpha, p_i, k)\Big],\end{aligned}$$

where the last equality comes from conditioning on the event $U(k)$.

The successive conditionings we did so far in the proof now allow us to tackle the probability of the event $W_i(k)$. The main point is that we reduced by conditioning the analysis within the event $U(k)$. The use of this fact is made precise in the coming proof. On the other hand, the event $Allnew(k)$ has actually been introduced only for the sake of convenience in the computations: handling the *independent* variables $\beta_j(k)$ is easier then the handling of the $B_j(k)$.

Lemma 9.5

$$\Pr\Big[W_i(k) \mid U(k), Allnew(k), B(\alpha, p_i, k)\Big] \geq 1/n.$$

Proof:

In $U(k)$, the event $W_i(k)$ is the same as the event $\{B_i(k)$ is (the unique) maximum among all the values $B_j(k)$ drawn by the participants of round $k\}$. One crucial consequence of this fact is that in $U(k)$, the event $W_i(k)$ depends only of the values of the local variables $B_j(k)$; $j = 1, \ldots, n$. Recall that $Allnew(k)$ represent the set of executions where in round k all the values of the local variables B_j and R_j of the participating processes are erased and replaced with new independent values drawn from the sequences β_j and ρ_j. On the other hand $B(\alpha, p_i, k)$ is a set of executions described in terms of conditions involving only the values of the local variables up to round $k-1$. These facts imply that

$$\Pr\Big[W_i(k) \mid U(k), Allnew(k), B(\alpha, p_i, k)\Big] =$$
$$\Pr\Big[W_i(k) \mid U(k), Allnew(k), \text{ } i \text{ participates in round } k\Big].$$

Now, the chances for process i to win in round k are minimal when the number of participating processes (the opponents of i) is maximal i.e. equal to n. This is expressed by

$$\Pr\Big[\, W_i(k) \mid U(k), \text{Allnew}(k) \,\Big] \geq \Pr\Big[W_i(k) \mid U(k), \text{Allnew}(k), M(k) = n\Big].$$

But in $U(k) \cap \text{Allnew}(k) \cap \{M(k) = n\}$ the sequence $(B_1(k), \ldots, B_n(k))$ is uniformly distributed; as a consequence the variable is maximum among all the variables $B_j(k); j = 1, \ldots, n$ with probability $1/n$. ■

We are now left with the task of finding a lower bound for $\Pr\Big[U(k) \mid \text{Allnew}(k), B(\alpha, p_i, k)\Big]$. But,

$$\Pr\Big[U(k) \mid \text{Allnew}(k), B(\alpha, p_i, k)\Big] = 1 - \Pr\Big[\neg U(k) | \text{Allnew}(k), B(\alpha, p_i, k)\Big],$$

and

$$\begin{aligned} &\Pr\Big[\neg U(k) \mid \text{Allnew}(k), B(\alpha, p_i, k)\Big] = \\ &\quad \Pr\Big[\neg U(k), B(k) = b \mid \text{Allnew}(k), B(\alpha, p_i, k), M(k) = m\Big] + \\ &\quad \Pr\Big[\neg U(k), B(k) < b \mid \text{Allnew}(k), B(\alpha, p_i, k), M(k) = m\Big]. \end{aligned}$$

We will estimate these two terms with the help of the two following lemmas.

Lemma 9.6

$$\Pr\Big[\neg U(k), B(k) = b \mid \text{Allnew}(k), B(\alpha, p_i, k)\Big] \leq 1/16$$

Proof: We will actually establish the following fact

$$\Pr\Big[\neg U(k), B(k) = b \mid \text{Allnew}(k), B(\alpha, p_i, k), M(k) = m\Big] \leq 1/16.$$

The result stated in the lemma will then follow by integration on m.

Let $i_1 = i, i_2 \ldots, i_m$ the processes having participated in the round. Note that these indices are random variables except $i_1 = i$ that is fixed by the conditioning on $B(\alpha, p_i, k)$. Then

$$\begin{aligned} &\Pr\Big[B(k) = b, \neg U(k) | \text{Allnew}(k), B(\alpha, p_i, k), M(k) = m\Big] \\ \leq\; &\Pr\Big[B(k) = b | \text{Allnew}(k), B(\alpha, p_i, k), M(k) = m\Big] \\ =\; &\Pr\Big[\cup_{j=1}^{m} \{B_{i_j} = b\} | \text{Allnew}(k), B(\alpha, p_i, k), M(k) = m\Big]. \end{aligned}$$

But, on $\{1, \ldots, b-1\}$, the law of $(B_{i_j}(k))_{j=1,\ldots,m}$ conditioned on $\text{Allnew}(k)$, is the law of $(\beta_j(k))_{j=1,\ldots,m}$ or of $(\beta'_j(k))_{j=1,\ldots,m}$. (Recall that a priori the $B_j(k)$ are not independent whereas the $\beta_j(k)$ are independent identically distributed variables.) This leads to:

$$\begin{aligned}
\Pr\left[\bigcup_{j=1}^{m}\{B_{i_j}(k)=b\}\;\middle|\;\text{Allnew}(k), B(\alpha,p_i,k), M(k)=m\right] &= \Pr\left[\cup_{j=1}^{m}\{\beta_j(k)=b\}\right]\\
&= 1-\Pr\left[\cap_{j=1}^{m}\{\beta_j(k)<b\}\right]\\
&= 1-\prod_{j=1}^{m}\Pr\left[\beta_j(k)<b\right]\\
&= 1-\left(1-\frac{1}{2^{\lceil\log n\rceil+4}}\right)^m\\
&\le 1-\left(1-\frac{1}{16n}\right)^m\\
&\le 1-\left(1-\frac{1}{16n}\right)^n \le \frac{1}{16}.
\end{aligned}$$

The third equality comes from the independence of the variables β_j, the fourth from the fact that they have the same law; the last inequality is obtained substituting 1 for n since the function $x \mapsto 1-(1-\frac{1}{16x})^x$ is monotonically decreasing on $[1,\infty[$. This last bound "is universal" in n whereas the approximation lim $\phi(x)$ used by Rabin is valid only for big n's. ■

We now turn to the other term necessary for the evaluation of

$$\Pr\left[\neg U(k) \mid \text{Allnew}(k), B(\alpha,p_i,k).\right]$$

Lemma 9.7

$$\Pr\left[B(k)<b, \neg U(k) \mid \text{Allnew}(k), B(\alpha,p_i,k)\right] \le 1/3.$$

Proof: As in the previous lemma, we will actually establish the slightly stronger result

$$\Pr\left[B(k)<b, \neg U(k) \mid \text{Allnew}(k), B(\alpha,p_i,k), M(k)=m\right] \le 1/3.$$

$\Pr\left[B(k) < b, \neg U(k) | \text{Allnew}(k), B(\alpha,p_i,k)\right] = \Pr\Big[$The highest value of $B_i,\ldots,B_{i_m}$ is attained by at least two values and this value is less then b | Allnew(k), $B(\alpha,p_i,k)\Big]$.

As before, calling $i_1 = i, i_2,\ldots,i_m$ the random indices of the processes participating in round k, we can use the fact that the conditional law of the $B_{i_j}(k), j=1,\ldots,m$ (conditioned on the event Allnew(k)) is the law of $\beta'_i,\ldots,\beta'_{i_m}$, and rewrite the preceding into:

$\Pr\Big[$ Max$\{\beta'_i,\ldots,\beta'_{i_m}\}$ is attained by at least two values and this value is less then $b\Big]$.

This last expression is upper-bounded by

$$\Pr\Big[\ \mathrm{Max}\{\beta'_i, \ldots, \beta'_{i_m}\} \text{ is attained by at least two values}\ \Big].$$

The nice thing about this last expression is that the implicit cut-off value of b in the range of the variables B_i and β_i does not exist anymore with the β'_i. Hence this expression is a function of m only (<u>not of b</u>) that we will denote $T(m)$.

To compute $T(j)$, we look at the flipping coin process realizing each β'_i. We will think of this process as a game built of a succession of "elementary flips" in the following way. If we start the game with m players we let a player participate in the next flip iff it has not yet obtained a 0. The game ends when there is only one player left or there are no players. In the first case there is only one winner (i.e., only one maximum value among the variables β'_j) while in the second case there is more then one maximum value since more than one player has participated in the last play.

Based on this game, $T(j)$ is the probability that there is more than one winner, conditioned on the fact that j players are left playing the next flips. From the argument above it is clear that $T(0) = 1$ and $T(1) = 0$.

The value of $T(j)$ can be computed by conditioning on the number i of people drawing 0 in the next play (so that $j - i$ are left still waiting for a zero). The probability that i players among j draw 0 during a flip is $\binom{j}{i} 2^{-j}$, hence $T(j)$ can be expressed as follows:

$$T(j) = \sum_{i=0}^{j} \binom{j}{i} 2^{-j} T(j-i).$$

We prove now by induction that $T(j) \leq 1/3$ for $j \geq 2$. We first find $T(2) = 1/3$ by simply solving the related equation for $j = 2$.

Working by induction, assume now proven that $\forall i < j\ T(i) \leq 1/3$. Observe that for $j \geq 2$

$$\frac{1}{3}\left[\binom{j}{j-1} + \binom{j}{j}\right] = \frac{j+1}{3} \geq 1.$$

Then,

$$T(j)(1 - 2^{-j}) = 2^{-j} \sum_{i=1}^{j} \binom{j}{i} T(j-i),$$

and,

$$\begin{aligned} T(j)(2^j - 1) &= \binom{j}{1} T(j-1) + \cdots + \binom{j}{j-1} T(1) + T(0) \\ &\leq \binom{j}{1} T(j-1) + \cdots + \binom{j}{j-1} T(1) + \binom{j}{j-1}\frac{1}{3} + \binom{j}{j}\frac{1}{3} \end{aligned}$$

$$\begin{aligned} &\leq \frac{1}{3}\sum_{i=1}^{j}\binom{j}{i} \\ &= \frac{2^j - 1}{3}. \end{aligned}$$

The first inequality comes from the remark above, the second from the induction hypothesis and the fact that $T(1) = 0$; the last equality comes from the development of $(1+1)^j$. This establishes that $T(j) \leq 1/3$ and finishes the induction. ∎

Combining all the previous results, we can now finish the proof of our main theorem.

$$\begin{aligned} \Pr\left[W_i(k)\right] &\geq \frac{.99}{n}\Pr\left[U(k)|\text{Allnew}(k), B(\alpha, p_i, k)\right] \\ &\geq \frac{.99}{n}(1 - \Pr\left[\neg U(k)|\text{Allnew}(k), B(\alpha, p_i, k)\right] \\ &\geq \frac{.99}{n}(1 - \frac{1}{16} - \frac{1}{3}) \sim \frac{.59}{n}. \end{aligned}$$

∎

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

Corollary 9.8 *For every normal adversary A, the probability that a trying processor i enters the critical region within (the first) ℓ rounds in which it participates is at least* $1-(1-c/n)^\ell$.

Proof: It is not hard to use Theorem 9.1 to prove that the probability i *fails* to enter the critical region in the any round in which it participates is at most $1 - c/n$. Proceeding by induction on ℓ, it is not hard to use this fact to prove that the probability i fails to enter the critical region within (the first) ℓ rounds in which it participates 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 9.9 *Rabin's algorithm satisfies the following correctness conditions:*

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

9.3 Appendix

The following claim was made at the beginning of the proof of theorem 2. The essence of it is simply that the law of $B_j(k)$ conditioned on $Allnew(k)$ is the same as the law of $\beta_j(k)$.

Claim 9.3 $\forall j = 1,\ldots,n \;\; \forall l = 1,\ldots,b$

$$\left[\Pr\left[B_j(k) \geq l \mid B(\alpha,p_i,k)\right] \leq \Pr\left[\beta_j \geq l\right]\right]$$
$$=$$
$$\left[\Pr\left[B_j(k) \geq l \mid \neg New_j(k), B(\alpha,p_i,k), M(k)=m\right] \leq \Pr\left[\beta_j \geq l\right]\right]$$

Proof: On $\{1,\ldots,b\}$, the law of $B_j(k)$ conditioned on $New_j(k)$, is the law of $\beta_j(k)$ which is independent of $B(\alpha,p_i,k)$. Hence $\Pr\left[B_j(k) \geq l \mid New_j(k), B(\alpha,p_i,k)\right] = \Pr\left[\beta_j(k) \geq l\right]$. This implies that

$$\begin{aligned}
\Pr\left[B_j(k) \geq l \mid B(\alpha,p_i,k)\right] &= \Pr\left[B_j(k) \geq l \mid \neg New_j(k), B(\alpha,p_i,k)\right] \Pr\left[\neg New_j(k) \mid B(\alpha,p_i,k)\right] \\
&\quad + \Pr\left[B_j(k) \geq l \mid New_j(k), B(\alpha,p_i,k)\right] \Pr\left[New_j(k) \mid B(\alpha,p_i,k)\right] \\
&= \Pr\left[B_j(k) \geq l \mid \neg New_j(k), B(\alpha,p_i,k)\right] \Pr\left[\neg New_j(k) \mid B(\alpha,p_i,k)\right] \\
&\quad + \Pr\left[\beta_j(k) \geq l\right] \Pr\left[New_j(k) \mid B(\alpha,p_i,k)\right].
\end{aligned}$$

Using this along with the equality

$$\Pr\left[\beta_j(k) \geq l\right] = \Pr\left[\beta_j(k) \geq l\right] \left(\Pr\left[\neg New_j(k) \mid B(\alpha,p_i,k)\right] + \Pr\left[New_j(k) \mid B(\alpha,p_i,k)\right]\right)$$

we get the following equivalences

$$\begin{aligned}
&\Pr\left[B_j(k) \geq l \mid B(\alpha,p_i,k)\right] \leq \Pr\left[\beta_j \geq l\right] = \\
&= \begin{array}{c} \Pr\left[B_j(k) \geq l \mid \neg New_j(k), B(\alpha,p_i,k)\right] \Pr\left[\neg New_j(k) \mid B(\alpha,p_i,k)\right] \\ \leq \\ \Pr\left[\beta_j(k) \geq l\right] \Pr\left[\neg New_j(k) \mid B(\alpha,p_i,k)\right] \end{array} \\
&= \Pr\left[B_j(k) \geq l \mid \neg New_j(k), B(\alpha,p_i,k)\right] \leq \Pr\left[\beta_j(k) \geq l\right]
\end{aligned}$$

■

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 10: October 18

Lecturer: Nancy Lynch *Scribe: Anna Charny*[15]

10.1 General Resource Allocation Problem

In the previous lectures we considered the mutual exclusion problem, i.e. at most one process was allowed to be in the critical region at one time. In other words this problem can be said to model access to one critical resource. This problem can be generalized by allowing at most k processes to be in the critical region together, or in other words by modeling access to k critical resources.

This leads to two possible ways of describing the problem: exclusion problem and resource allocation problem.

10.1.1 Problem Description

1. *Exclusion problem:* The problem is presented as a set of "bad sets" of processes. By a "bad set" we mean a set of processes which are not allowed to enter the critical region simultaneously. The set of all "bad sets" is called an *exclusion set.* Obviously, if a particular "bad set"of processes belongs to the exclusion set, then any superset of this "bad set", obtained by adding any number of other processes to it, will still be a "bad set" and will belong to the exclusion set. In general, any set of collections of processes closed under containment defines some exclusion problem.

   Example 1: Mutual exclusion can be defined as

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

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

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

   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 $p_1, \ldots, p_n$.

[15]Based on lecture notes from 1988 scribed by Atul Shrivastava

2. *Multiple Resource problem:* The problem 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.

   $$\begin{array}{lcl} p_1 & : & R_1 \cap R_2 \\ p_2 & : & R_1 \cap R_3 \\ p_3 & : & R_2 \cap R_4 \\ p_4 & : & R_3 \cap R_4 \end{array}$$

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

Note: A resource problem can be converted into an 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 it is 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 case of exclusion problem, and is generally presented as a *resource allocation* problem.

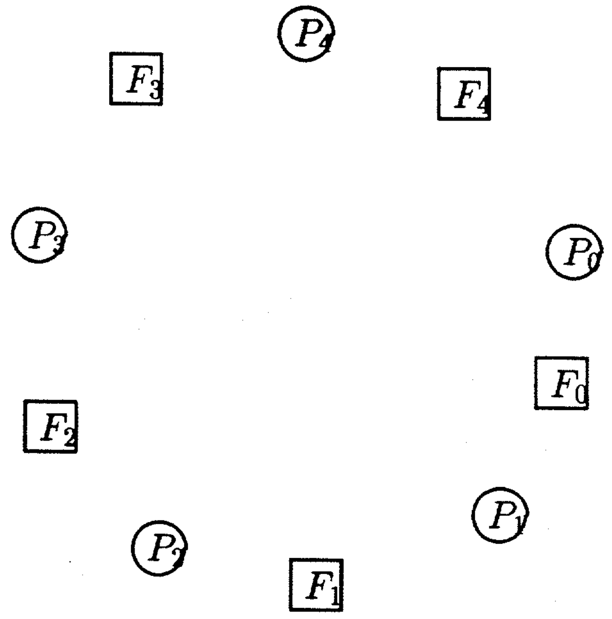


Figure 10.1: Dining Philosophers problem (n = 5)

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

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.

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

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

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 avoids busy-waiting by checking if a particular condition on the shared variable is satisfied before giving a turn to a particular process. If this condition is not satisfied, the process is put to sleep and is added to some queue (a set of some kind). When the value of the shared variable changes, the scheduler checks the queue for the processes whose conditions are now satisfied, and then wakes those processes.

10.3 A simple approach that deadlocks

The algorithm that we analyze here is presented in Figure 10.2.

The algorithm is conceptually simple, each process grabs the left fork first and then picks up the right fork. After getting both forks, it goes to 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 go to 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 goes to 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.

Shared variables:

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

Code for p_i:

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

  ** Critical Region **

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

  ** Remainder Region **

lock;
```

Figure 10.2: A simple solution

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 processes starve forever and the system cannot make any progress (i.e. no process can change regions).

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 by p_{i-1}, p_i and p_{i+1}; *sem* is an array of binary semaphores, initialized to 0.

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

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.

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

To prove progress we assume an infinite deadlocking execution in which some processes are in T or E, but no region change occurs.

The only impediments to the progress of any process are the P and V operations. Hence, it is enough to show that 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 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 $\mathbf{TEST}(i)$, 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 $\mathbf{TEST}(i)$, $control(i) \leftarrow 2$ 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 processes 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 $\mathbf{TEST}(i)$ to have failed. Otherwise, $V(sem(i))$ would have been executed. Therefore,

Shared variables:

> *mutex* is a binary semaphore, initially 1
> $\forall i, control(i) \in \{0,1,2\}$, written by and read by several processes,initially 0
> $\forall i, sem(i)$ 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

$$\Rightarrow (control(i-1) = 2 \vee control(i+1) = 2)$$

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

The algorithm allows individual process to get locked out. Consider a situation 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.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 11: October 18

Lecturer: Nancy A. Lynch *Scribe: Boaz Patt*

We continue studying the problem of resource allocation, focusing on the Dining Philosophers. There are several issues by which we evaluate a solution to the Dining Philosophers problem:

- Distributed solutions are considered "better" than centralized solutions (Dijkstra's algorithm is centralized: the algorithm uses a global mutual exclusion variable accessed by all processes).
- Using read/write variables is considered better than using test-and-set variables, but one can implement test-and-set variables with read/write variables.
- We would like to ensure that there is no lockout, i.e., every philosopher who wishes to eat is guaranteed to eventually eat, or even better:
- We would like to bound the time that may take for a philosopher from the point he gets hungry until he eats.

11.1 Symmetric algorithms: an impossibility result

An interesting class of algorithms is the class of *symmetric* algorithms. An algorithm is said to be symmetric if all processes execute the same code (processes have no identifiers, and may refer only to "the process on the left", or "fork on the right", etc.). As for Dining Philosophers, we have the following theorem.

Theorem 11.1 *There is no symmetric deterministic solution to the Dining Philosophers problem.*

Proof: Let A be a symmetric algorithm for n processes. Refer to figure 11.1. Let process p_0 take its first step. If it accesses the variable on its right (F_0), let the next step be taken by p_1, and then $p_2, \ldots, p_{n-1}$; if it accesses the variable on its left, let $p_{n-1}, p_{n-2}, \ldots, p_1$ take the next steps. Since all p_i act according to the same deterministic code, and since all the variables they can access have the same values, by the end of these n steps all the processes are in the same internal state. We may proceed in this fashion by induction, maintaining the invariant that by the end of every such "round" of n steps, all processes are in the same state, and all shared variables have the same value. But this fact, when coupled with the progress property, violates the exclusion property: if any process is in the Critical Region

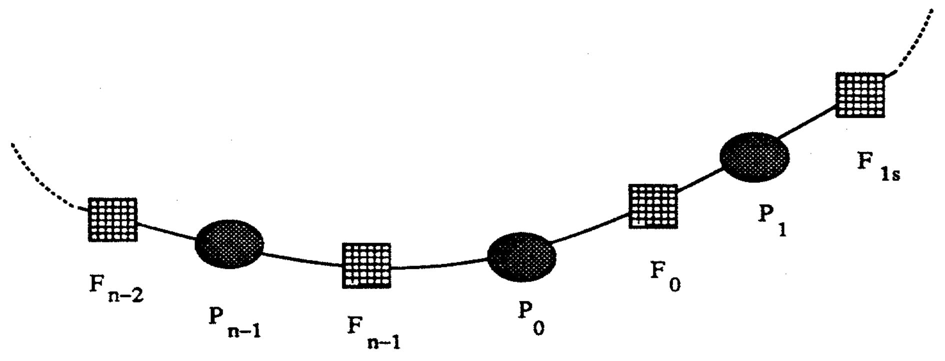


Figure 11.1: Dining Philosophers

by the end of a round, then *all* processes are in the Critical Region by the end of this round. Hence a symmetric deterministic algorithm must violate either the exclusion or the progress property. ■

The proof indicates a fundamental problem in computing on a ring network: we need to "break" the symmetry among the processes. The two ways for doing so are either using the unique identifier we assume each process to have (which really means that we assign different codes for different processes), or employing non-deterministic algorithms (that is, using randomness). We'll see examples for both approaches.

11.2 Left-Right Algorithm (Burns)

The following algorithm is due to Burns (but is almost a part of the distributed computation folklore). It ensures exclusion, progress, and it prevents lockout — with worst case time bound of a constant independent of n.

This latter property is of special importance. Analyzing the possible executions of any algorithm solving Dining Philosophers, we find that there may be a state in which p_{i_1} is waiting for a resource (fork) currently held by p_{i_2}, while p_{i_2} in turn waits for a resource p_{i_3} is currently holding, and so on. We call such a sequence of processes $p_{i_1} \rightarrow p_{i_2} \rightarrow p_{i_3} \rightarrow \cdots$ a *waiting chain.* Since the processes in a waiting chain enter the Critical Region sequentially, when we want to minimize the bound on the time for a process to be in the Trying Region, we should strive to bound the maximal length of such waiting chain.

In the LR algorithm (fig. 11.2), we assume that the processes are numbered in successive order; we further assume that we have test-and-set shared variables with a FIFO access queue. For simplicity, we assume here that the number of processes is even. There are two programs: one for the processes with odd numbers, and one for those with even numbers.

Program for process $2i$:

```
    wait for $F_{2i-1}$
    wait for $F_{2i}$
    ** Critical Section
    release $F_{2i-1}$ and $F_{2i}$ in any order
    ** Remainder Region
```

Program for process $2i+1$:

```
    wait for $F_{2i+1}$
    wait for $F_{2i}$
    ** Critical Section
    release $F_{2i}$ and $F_{2i+1}$ in any order
    ** Remainder Region
```

Figure 11.2: The Left-Right algorithm

The basic strategy is very simple: odd-numbered processes seek their left fork first, and even-numbered processes seek their right fork first. The forks are implemented as a binary flag that indicates the status of the fork (occupied/free), and a queue to keep track of the FIFO order of requests (which may arrive from only 2 processes).

11.2.1 Properties

Exclusion is immediate from the code and the fact that the variables are test-and-set variables. We will show that there is no lockout by proving an explicit time bound on the time for a process to reach the Critical Region.

The key idea in the algorithm above is that a fork between two processes is, for both processes, either the first or the second one they wish to get. This implies that the maximal length of a waiting chain is bounded by 2, and gives the following nice upper bound.

Theorem 11.2 *Suppose that any step time is bounded by s, and that the time spent in the Critical Region by any process is bounded by c. Then the time any process spends in the Trying Region (until it gets to the Critical Region) is no more than $3c + 15s$.*

Proof: Denote

T = worst-case time from entering the Trying Region until entering the Critical Region,
S = worst-case time from having one fork until entering the Critical Region.

We bound T by case analysis. Consider a process p_i entering the Trying Region. Its first step is trying to get the first fork. If it succeeds (after time s), then after no more than additional

S time p_i enters the Critical Region. Otherwise, its neighbor has the fork, and as mentioned above, it must be its first fork, too. Hence the time until the neighbor releases this fork is bounded by $s + S + c + 2s = 3s + c + S$ (getting the fork, reaching Critical Region, Critical Region, releasing two forks). It follows that in this case, p_i enters the Critical Region after at most $s + (3s + c + S) + s + S = 5s + c + 2S$ time. We conclude that

$$T \leq \max\{s + S,\ 5s + c + 2S\} = 5s + c + 2S \tag{11.1}$$

Let us now bound S. Suppose a process is requesting the second fork. The time for the case where it gets the fork immediately is bounded by s. Otherwise, its neighbor has the fork, and it is the neighbor's second fork. Hence, the time until the second fork is released in this case is bounded by $s + c + 2s$, and the time until the process enters the Critical Region is bounded by $5s + c$ (adding s for the request and s for getting the fork). We conclude that

$$S \leq \max\{s,\ 5s + c\} = 5s + c \tag{11.2}$$

Combining (11.1, 11.2) yields

$$T \leq 15s + 3c$$

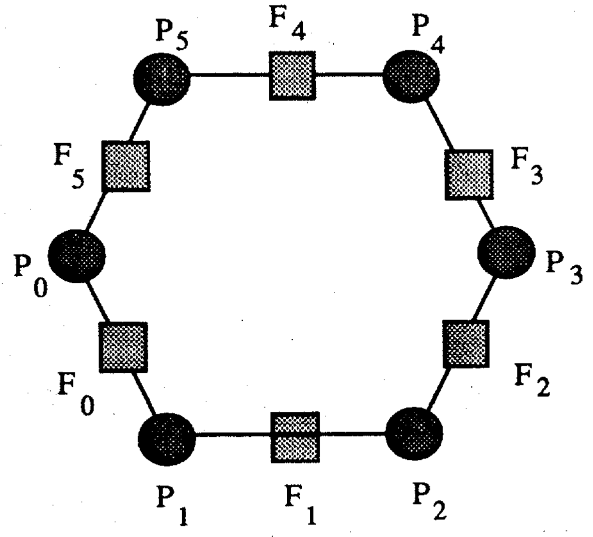
■

Remark. The independence of n is a good property for a distributed algorithm — it may be thought of as network transparency. But note that the RL algorithm relies heavily on the static knowledge of the parity of the identifiers of the processes. This assumption might not be reasonable in a truly distributed networks.

11.3 Generalizing LR — Lynch's method

An obvious question is how to generalize the idea of minimal-length waiting chains to a general resource-allocation problem. Lynch's method solves the exclusion problems that can be expressed by conjunctions. Denote the set of resources by $R = \{r_1, \ldots, r_m\}$. We assume that every process p_j requires $\bigwedge_{i \in I_j} r_i$, where I_j is a subset of R (this means that there are no alternatives: a process must get a certain fixed subset of the resources).

We associate with every resource r_k a queue q_k in which the processes are waiting for the resource, and a binary flag s_k indicating whether r_k is in use. We say that p_i *waits for* p_j if p_j has some resource r_k, and p_i is in r_k's queue. The basic strategy will be *hierarchical resource allocation*. In this strategy, the resources are totally ordered, and the processes seek the resources they need in an increasing order. It is easy to see that this strategy ensures progress, as well as no lockout: the first process on the final queue is never blocked, and no process ever relinquish. The problem now is how to define a total ordering that minimizes the length of the potential waiting-chains.



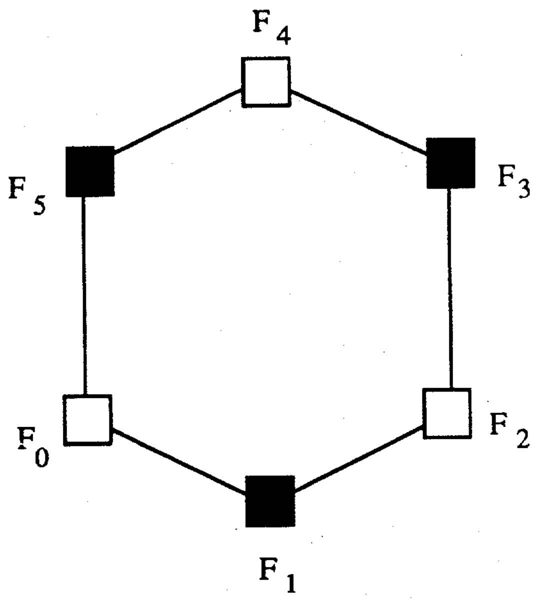


Figure 11.3: 6 dining philosophers and their corresponding resource graph

11.3.1 Constructing a "good" total ordering

1. Construct the *resource allocation graph* $G = (V, E)$, where

$$\begin{aligned} V &= \{r_0, \ldots, r_{n-1}\} \text{ (the set of resources)} \\ E &= \{(r_i, r_j) : \text{ there is a process needing } r_i \text{ and } r_j\} \end{aligned}$$

2. Color the nodes of the graph — that is, assign each $r \in V$ a color $c(v)$ in a way such that if r_i and r_j are adjacent, then $c(r_i) \neq c(r_j)$. We would like to minimize the number of colors needed. Obtaining the minimal number is NP-hard, but a "low" number of colors will do.

3. Order the *colors* totally in an arbitrary way. This induces a partial order on the resources ($r_i < r_j$ if $c(r_i) < c(r_j)$).

4. Complete the partial order to a total order (by topological sorting).

Example. Consider once again the Dining Philosophers. The colored resource allocation graph is shown in figure 11.3. Note that the solution reduces to the Left-Right algorithm.

The above strategy bounds the maximal length of a waiting chain by the number of distinct colors (in the Dining Philosophers, this reduces to the LR algorithm, where the number of colors is 2). This follows from the fact that if a process p is waiting for a resource held by another process p', then p' may be waiting only for a resource of a "higher" color.

11.3.2 Time analysis

Lynch's resource allocation strategy gives a time bound that is not directly dependent on the total number of processes and resources, but rather on a "local" parameters, as the following theorem states. This constitute an improvement on the naive solution where the resources are totally ordered arbitrarily: although this approach ensures progress, it may result in waiting time linear in the number of processes.

Theorem 11.3 *Let*
s be an upper bound on a step time,
c be an upper bound on the Critical Region time,
k be the number of colors, and
m be the maximal number of users for a single resource.
Then the worst case time for a user to get into the Critical Region is $\left((m^k)c + (km^k)s\right)$.

Note that the bound is not proportional to the k (length of a longest waiting chain), but it is actually exponential in k. This bound can be approached by complex interleavings (as suggested by the proof below).

Proof: The general outline is as follows. Define $T_{i,j}$ to be the worst case time from when a process reaches position j from the front of the queue for resource i, until it reaches the Critical Region. Let $0 \leq i \leq k-1$, and $0 \leq j \leq m-1$, where position 0 is interpreted as actually having the resource. We wish to bound $T \leq T_{0,m-1}$. This can be done by setting up equations as we did for the LR algorithm.
The base case is when a process is not waiting for any other resource:

$$T_{k-1,0} = 0$$

Otherwise, either the process waits for the process ahead to get the resource and to give it up:

$$T_{i,j} \leq T_{i,j-1} + c + ks + T_{i,0}$$

or, the process gets a resource and wait the next resource it needs:

$$T_{i,0} \leq s + T_{i+1,m-1} \ .$$

Clearly, the worst case is when all m processes need all k resources. The time for a process to reach the Critical Region is bound by $T_{0,m-1}$. Now, for all $0 \leq i < k-1$

$$\begin{aligned} T_{i,m-1} &\leq m(c+ks) + mT_{i,0} \\ T_{i,0} &\leq s + Ti, m-1 \end{aligned}$$

and $T_{k-1,0} = s$. Assuming $s \ll m(c+ks)$ and that $m \gg 1$, we obtain

$$\begin{aligned} T_{0,m-1} &\leq (c+ks)\sum_{i=0}^{k-1} m^i \\ &\leq m^k(c+ks). \end{aligned}$$

■

11.3.3 Open questions

The following question remains open. Can a solution of this kind be given for some more general resource allocation problem, where the generalization is either

1. Allowing arbitrary static constraints (i.e., disjunctions in the problem formulation), or
2. Allowing the environment to be dynamic, i.e., when the processes and resource requests may change during the execution.

Some of these questioned are melt with in later lectures: see the "Drinking Philosophers".

11.4 Rabin-Lehmann Algorithm

The Rabin-Lehmann algorithm we consider next is a randomized algorithm that ensures exclusion and progress (there exists a modification of it – that will not be given – which ensures no lockout). All processes are identical — the symmetry breaking is achieved by randomization.

The notion of adversary considered here is different from the one considered for Rabin's mutual exclusion (ME) algorithm. The ME adversary depended on the history of *region changes*, and did not have access to the states of processes. This time, the adversary has more power: he knows the complete past *execution*, including states and past random draws. The indivisible actions are individual fork accesses. In the following discussion, we denote

$$opp(s) = \begin{cases} L, & \text{if } s = R \\ R, & \text{if } s = L\,. \end{cases}$$

An informal description of the procedure is "choose randomly a side in each iteration. Wait for the fork in the chosen side, and after getting it, just check *once* for the second. If succeeded, proceed to the Critical Region. Otherwise, put fork down and try again with new random choice."

Code for process i:

```
do forever
    $s \leftarrow random$                  /* choose $R$ or $L$ with equal probability */
    wait until $s$ is free
    pick up $s$
    if $opp(s)$ is free then
        pick up $opp(s)$; goto L
    else put $s$ down

L:
** Critical Region
put down both forks
** Remainder Region
```

Figure 11.4: The Rabin-Lehmann algorithm

11.4.1 Correctness

Exclusion is immediate from the indivisibility of the individual fork accesses. As for progress, we formalize the notion of an adversary.

An adversary $\mathcal{A}$ is a function mapping finite executions to the set of processes, determining the next process to take step. The adversary is *fair* if only fair executions are generated. More precisely, denote by $exec(\mathcal{A}, D)$ the unique execution generated by adversary $\mathcal{A}$ and a sequence of random draws D. $\mathcal{A}$ is fair if $exec(\mathcal{A}, D)$ is fair for all D. By the assumption that R and L are drawn with probability 1/2, there is a probability associated with each (set of) sequence of draws. Thus, given an adversary $\mathcal{A}$, we have a probability distribution on the set of executions generated by $\mathcal{A}$. Having this, denote by Z the set of fair executions with no progress, i.e., the set of executions in which there is some prefix α such that after α there is someone in the Trying Region and no change of region occurs. Our goal is to show that Prob [Z] = 0. We proceed as follows.

Remark 11.4 *In any execution in Z, there are infinitely many fork pickups.*

Proof: Assume not. Then after the last fork pickup, all processes that are stuck in the Trying Region must eventually put their forks down, and then nothing can stop additional pickups. ■

Remark 11.5 *If p chooses infinitely often, then p chooses R infinitely often and L infinitely often with probability 1. Formally, denote by C_p the event that p chooses infinitely often, by L_p (R_p) the event that p chooses L (respectively, R) infinitely often. Then* $\text{Prob}[L_p|C_p] = \text{Prob}[R_p|C_p] = 1$.

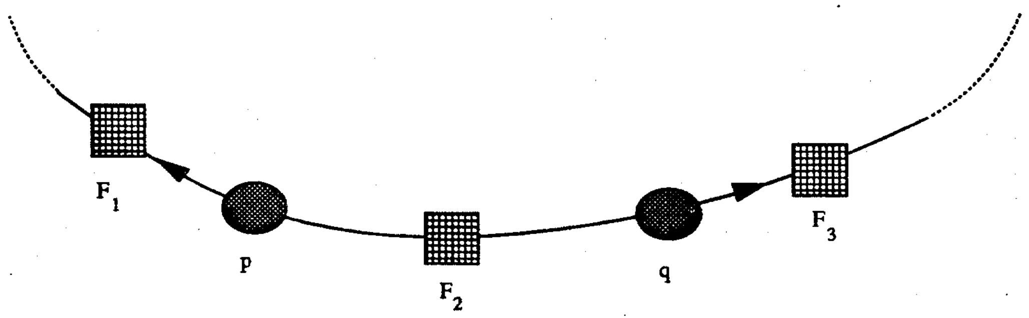


Figure 11.5:

Proof: Let $S = s_1, s_2, \ldots$ denote the infinite sequence of choices made by process p. It suffices to prove that Prob $[|\{i : s_i = L\}| \leq k] = 0$ for all k. We compute this probability as follows.

$$\begin{aligned}
\text{Prob}\,[|\{i : s_i = L\}| \leq k] &= \lim_{n\to\infty} \text{Prob}\,[|\{i : s_i = L,\ i \leq n\}| \leq k] \\
&= \lim_{n\to\infty} \sum_{m=0}^{k} \binom{n}{m} 2^{-n} \\
&\leq \lim_{n\to\infty} n^k 2^{-n} \\
&= \lim_{n\to\infty} 2^{-n+k\log n} \\
&= 0\,.
\end{aligned}$$

■

Lemma 11.6 *Let p and q be neighbors. If p picks up fork infinitely often and q does not, then p eats infinitely often with probability 1. Formally, denote by P_p the event that p picks up fork infinitely often, and by E_p the event that p eats infinitely often. Then* Prob $[E_p|P_p] = 1$.

Proof: Without loss of generality, assume p is left of q (figure 11.5). If p picks up infinitely often, then it chooses infinitely often, and hence, by remark 11.5, it chooses L infinitely often with probability 1. This implies that p eventually succeeds getting F_1 each time, or else it would get stuck, contradicting the assumption of infinite pickups. Having F_1, p looks for its right fork, namely F_2, which by the assumption, starting from some point in the execution, is never taken by q. Thus p, from some point of the execution and on, always succeeds in getting F_2, and eats infinitely often. ■

Lemma 11.7 *If an execution makes no progress, then with probability 1 every process picks up a fork infinite number of times. Alternatively:* Prob $\left[\bigwedge_p P_p | Z\right] = 1$.

Proof: By remark 11.4, there are infinitely many fork pickups. Assume that some process q does not pick up a fork infinitely often. Then by lemma 11.6, its neighbors eat infinitely often, contradicting the assumption that the execution does not progress. ■

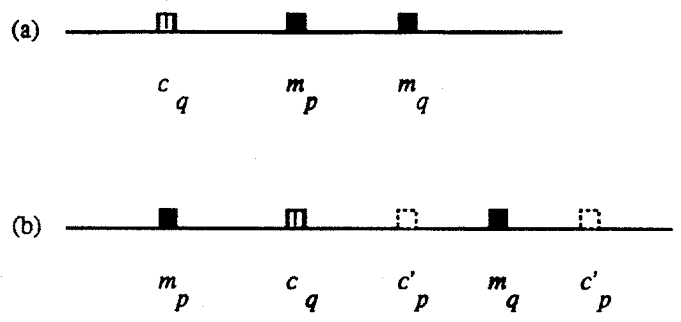


Figure 11.6: Possibilities after a good execution

Now call a finite execution *good execution* with respect to processes p and q, if p's last random choice was L and q's last random choice was R (see figure 11.5). The reason of calling such executions good is shown in the following lemma.

Lemma 11.8 *Let α be a good execution. If every process picks up infinitely often after α, then either p or q eats after at most two additional random draws (of p and q) with probability at least* $1/2$.

Proof: Assume, without loss of generality, that q made his last random choice after p's last random choice. Denote by m_p and m_q the points in the execution in which p and q examine their second fork (F_2), respectively. Denote the point of the execution in which q makes his choice c_q. Consider the execution at c_q. There are the following alternatives.

(a) If m_p has not occurred (figure 11.6 a), then whichever examines F_2 first (after c_q) eats (in the execution depicted in figure 11.6 a, p eats).

(b) If m_p has already occurred (figure 11.6 b), then either p succeeded getting its second fork (and then we are done), or else p will choose again. With probability $1/2$ p chooses L again. At this point (c'_p in figure 11.6 b) either q had already examined F_2 (and we are done), or else whichever examines F_2 first eats (as in case (a)).

■

Remark 11.9 *If p and q are neighbors that choose infinitely often, then infinitely many of the prefixes of the execution are good.*

Proof: Similar to proof of Remark 11.4. ■

We may now conclude with the following theorem.

Theorem 11.10 Prob [Z] $= 0$.

Proof: Suppose by the way of contradiction that Prob [Z] > 0. Then we may condition probabilities on Z. Let p and q be neighbors as in figure 11.5. By Lemma 11.7, Prob $[P_p \wedge P_q | Z] = 1$, and hence, by Lemma 11.8, Prob [Progress$|$Z] $= 1$, a contradiction. ■

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 12: October 30

Lecturer: Isaac Saias *Scribe: Marc LeBlanc*[16]

The first section of this course was devoted to shared-memory algorithms. In this lecture we will discuss **network algorithms**, designed for groups of processes that communicate via messages (as opposed to communicating via shared variables). This model is referred to in the bibliography as the **message-passing** model of communication.

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.

12.1 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.

[16]Based on lecture notes from 1988 scribed by Mike Einsenberg and Magda Nour

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).

12.1.1 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).

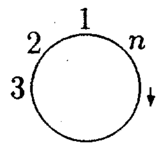


Figure 12.1: The worst case identifier ordering for the Le Lann-Chang-Roberts algorithm.

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 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 12.1. In this case, each identifier i is passed approximately i times, so the total number of messages is

$$\sum_{i=1}^{n} i = O(n^2)$$

In the best case scenario, the process identifiers are arranged in the opposite order, as in Figure 12.2. 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 $O(n)$.

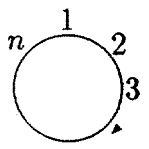


Figure 12.2: The best case identifier ordering for the Le Lann-Chang-Roberts algorithm.

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 Rabin's randomized 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:

$$\binom{n-1}{j-1} + \binom{n-2}{j-1} + \dots \binom{j-1}{j-1} = \binom{n}{j}$$

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:

$$n + \frac{n}{2} + \frac{n}{3} + \ldots \frac{n}{n} = n(1 + 1/2 + \ldots 1/n) = O(n \log n)$$

12.1.2 Hirshberg-Sinclair's 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.
- 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 12.3.

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

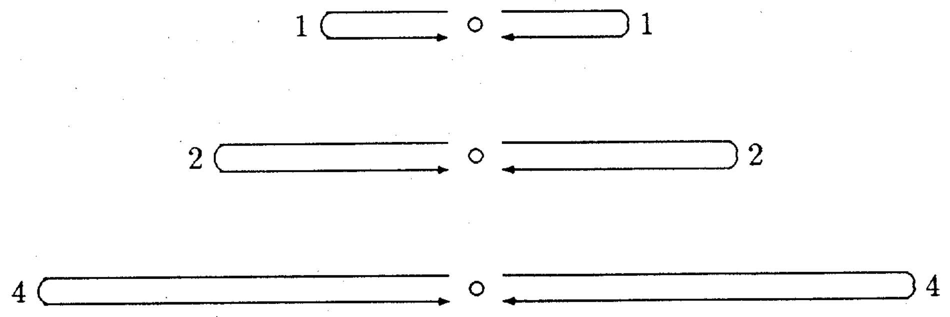


Figure 12.3: Successive message-sends in the Hirshberg-Sinclair algorithm

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

$$4\Big((1*n)+(2*\left\lceil\frac{n}{2}\right\rceil)+(4*\left\lceil\frac{n}{3}\right\rceil)+\ldots(2^i*\left\lceil\frac{n}{2^{i-1}+1}\right\rceil)+\ldots\Big)$$

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 $4n$.

12.2 Peterson's 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.
- 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.

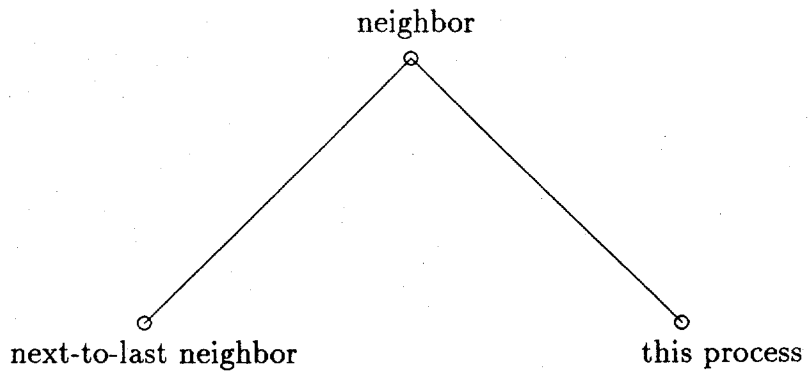


Figure 12.4: A "good" configuration for a Peterson-algorithm process.

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 12.4. 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(next-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.

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 12.5.

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

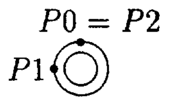


Figure 12.5: The last phase of the Peterson algorithm. $P0$ is the winner.

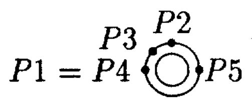


Figure 12.6: The next-to-last phase of the Peterson algorithm.

at worst only extend as far as $P1$, as depicted in Figure 12.6. 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 12.7.

12.2.1 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 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

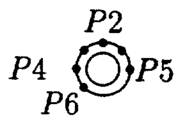


Figure 12.7: The next preceding phase of the Peterson algorithm.

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 12.1 *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 12.8. 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

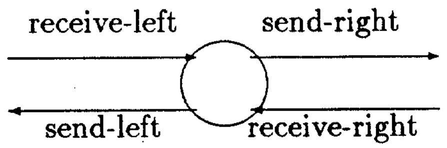


Figure 12.8: A process participating in a leader election algorithm.

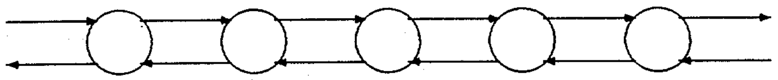


Figure 12.9: A line of leader-electing automata.

the automata behave when arranged *not* in a ring, but simply in a straight line, as in Figure 12.9. 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 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 12.10.

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 12.2 *For every $i \geq 0$, there is an infinite set of disjoint lines, $\mathcal{L}_i$, such that for all $L \in \mathcal{L}_i$, $|L| = 2^i$ 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

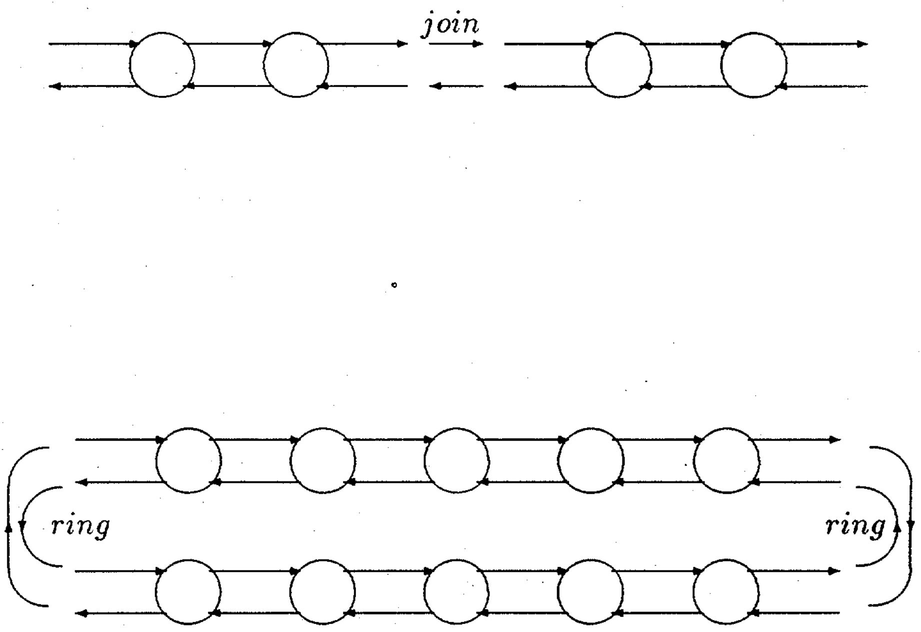


Figure 12.10: Join and ring operations.

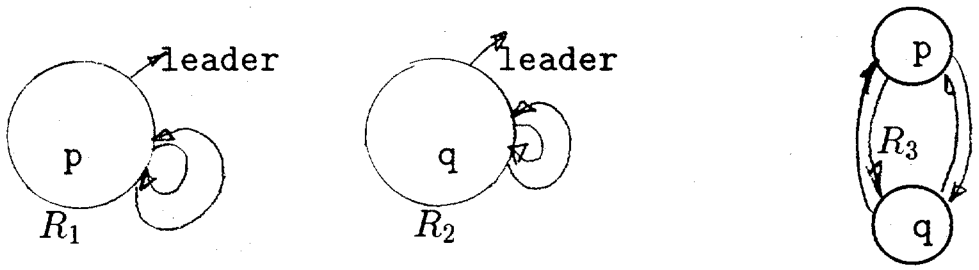


Figure 12.11: Basis for proof of Lemma 15.1

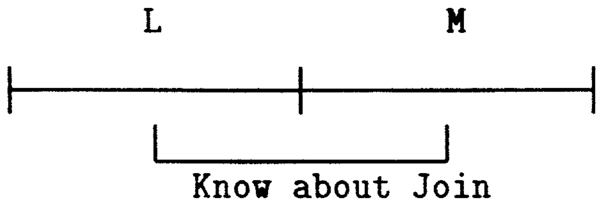


Figure 12.12: Join(L,M)

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 12.11.

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.

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 12.3 *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 12.12. 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 processes that know about each join extend at most half way into either segment, these messages will be non-interfering. Similarly for R_2.

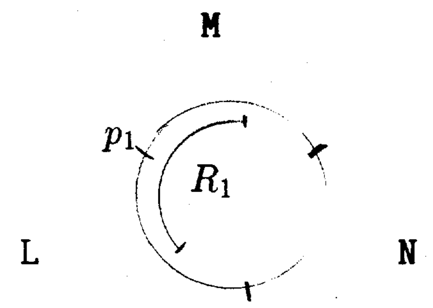


Figure 12.13: Join(L,M,N):Case 1

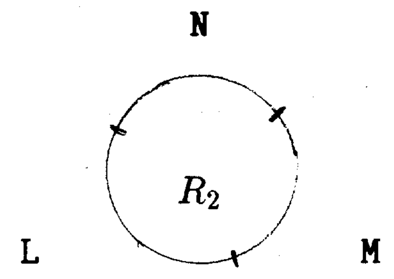


Figure 12.14: Join(L,N,M)

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 12.13. We consider cases based on the position of p_2 in R_2 (Figure 12.14) to find a contradiction.

- Case 1: p_2 is between the midpoint of L and the midpoint of N as in Figure 12.15

  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 12.16. Then it also occurs in R_2 and gets elected there too as in Figure 12.17. 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 12.18.

■

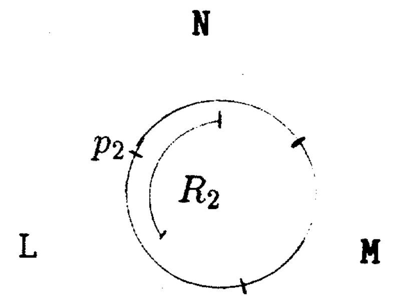


Figure 12.15: Join(L,N,M)

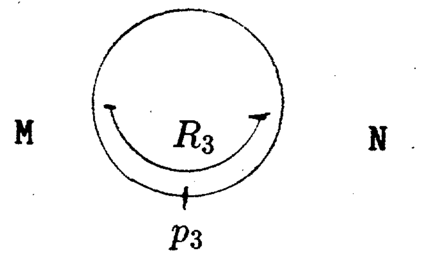


Figure 12.16: Join(M,N): Leader elected in the lower half

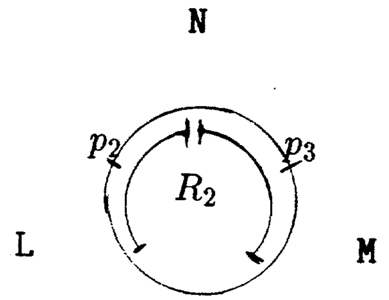


Figure 12.17: Join(L,N,M)

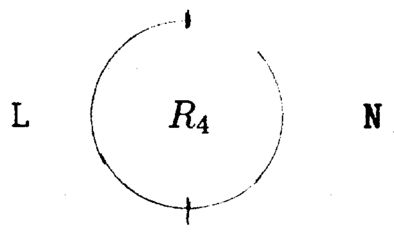


Figure 12.18: Join(L,N): Leader elected in the lower half

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.

6.852 Distributed Algorithms	**Fall Semester, 1990**
Lecture 13: October 25	
Lecturer: Nancy Lynch	*Scribe: S.J. Lee*[17]

13.1 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 13.1. The index i on the read

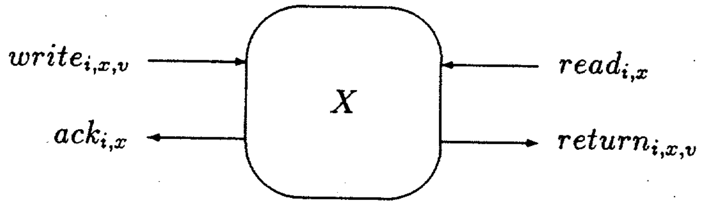


Figure 13.1: Concurrent Read/Write Register X

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.

13.1.1 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

[17]Based on the complete lecture notes from 1988 scribed by Sanjay Ghemawat

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

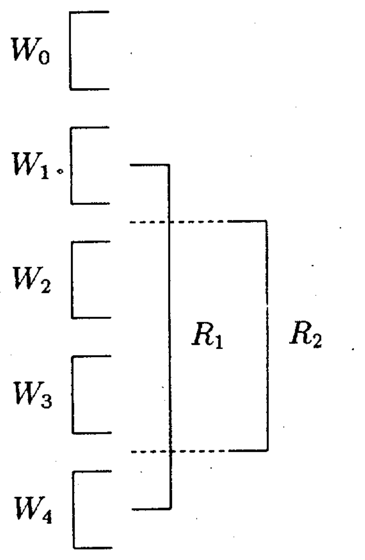


Figure 13.2: Read Overlapping Writes

Figure 13.2. 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.

13.2 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 13.3 should be obvious.

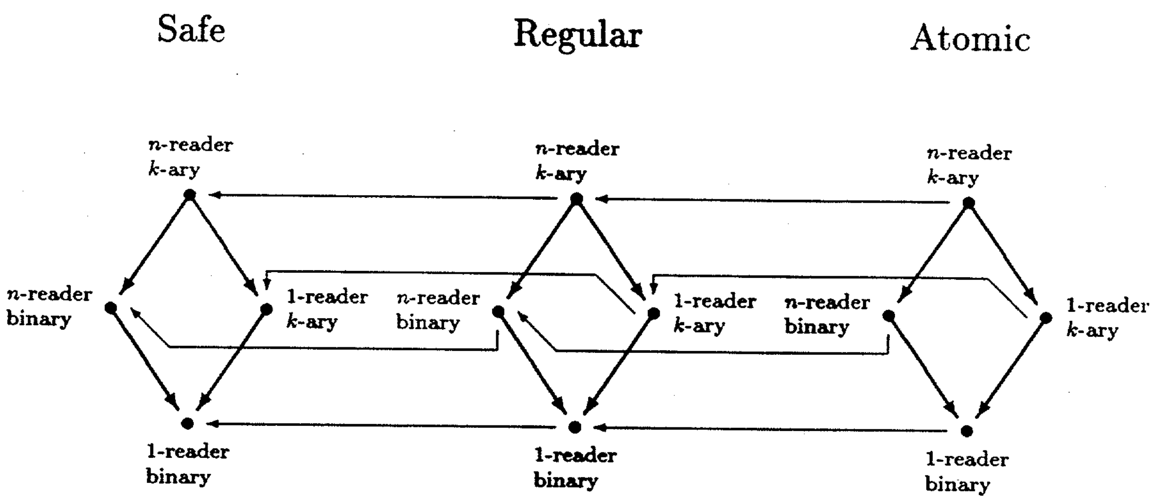


Figure 13.3: Obvious implementation relationships between register types. An arrow from type A to type B means that B can be implemented using A

13.3 Register Constructions

Lamport presents five constructions to show other implementation relationships. All of these constructions have a similar flavor. For example, consider Figure 13.4. 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, ext(write) and ext(read) denote external operations whereas int(write) and int(read) denote internal operations.

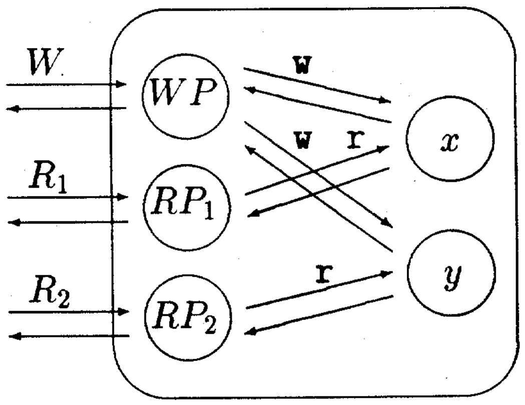


Figure 13.4: Example: Implementing a 2-reader register with two 1-reader registers

13.3.1 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 13.5. Read process i is connected to the read line of the ith physical register.

Code for *ext*(write)(v)

For all i in $\{1,\ldots,n\}$, send *int*(write)(v) to x_i. Wait for *int*(acks) from all x_i and then send *ext*(ack). The *int*(writes) can be done either concurrently, or in any order.

Code for *ext*(read)$_i$

Send *int*(read) to x_i. Wait for *int*(return)(v) and then send *ext*(return)(v).

Claim 13.1 *If $x_1,\ldots,x_n$ are safe registers, then so is the logical register.*

Proof: Within each *ext*(write), for any particular x_i, exactly one *int*(write) is performed on that register. Therefore, since *ext*(write) operations occur sequentially, *int*(write) operations for a particular x_i are also sequential. In addition, *int*(read) operations for a particular x_i are also sequential. Therefore, each physical register has the required sequentiality of accesses.

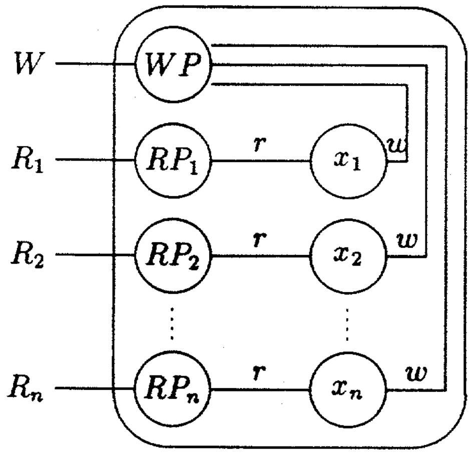


Figure 13.5: N-Reader Registers from n 1-Reader Registers

If a *ext*(read), say by RP_i, does not overlap any *ext*(write), then its contained *int*(read) does not overlap any *int*(write) to x_i. Therefore, safety of x_i assures that the *int*(read) operation gets the value written by the last completed *int*(write) to x_i. This is the same value as written by the last completed *ext*(write), and since RP_i returns this value, this *ext*(read) returns the value of the last completed *ext*(write). ■

Claim 13.2 *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 *ext*(read) which does not overlap a *ext*(write) returns the correct value. Therefore, we only need to show that if a *ext*(read) R overlaps some *ext*(write) operations, then it returns the value written by a feasible *ext*(write). Since the *int*(read) r for R falls somewhere inside the duration of R, the set of feasible *int*(writes) for r corresponds to a subset of the feasible *ext*(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 *ext*(writes). ■

Claim 13.3 *This construction does not make the logical register atomic even if the x_i are atomic.*

With this construction, Figure 13.3 reduces to Figure 13.6.

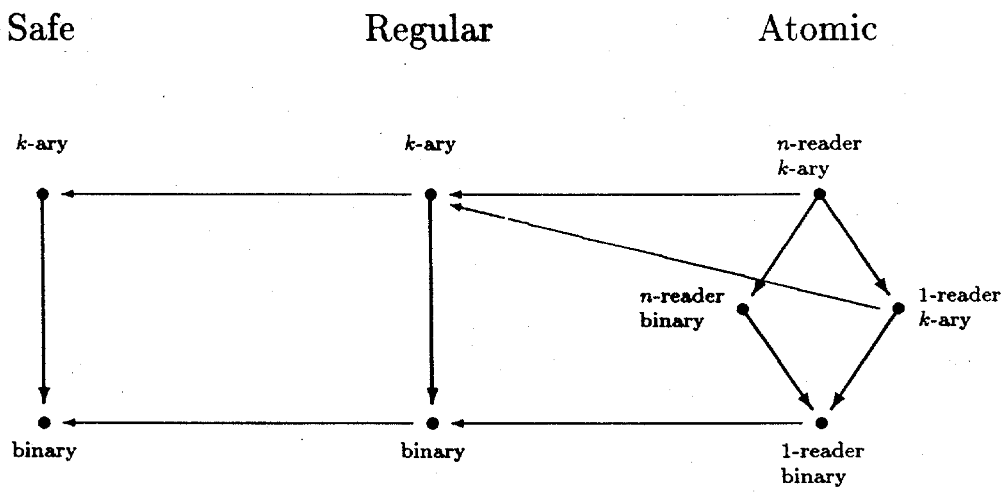


Figure 13.6: Collapsed Implementation Relationships

13.3.2 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.

13.3.3 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 13.7).

Code for ext**(write)**(v)

For i in $\{1, \ldots, l\}$ (any ordering), write bit i of the value to register x_i.

Code for ext**(read)**$_i$

For i in $\{1, \ldots, l\}$ (any ordering), read bit i of value v from register x_i. $ext(\text{return})(v)$.

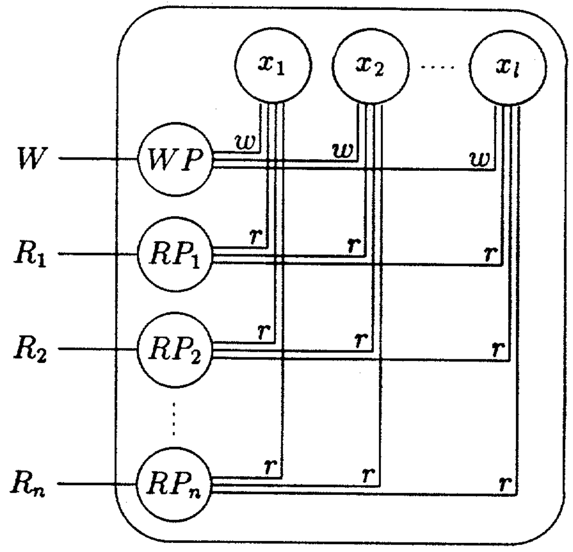


Figure 13.7: K-ary Safe Registers from Binary Safe Registers

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 13.6 reduces to Figure 13.8. .

13.3.4 Binary Regular Register from Binary Safe Register

A binary regular register can easily be implemented using just one binary safe register (see Figure 13.9). 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 *int*(write) only when it gets a *ext*(write) which would actually change the value of the register. If the *ext*(write) is just rewriting the old value, then WP finishes the operation right away without touching x. Therefore, all low level *int*(writes) toggle the value of the register. Now consider the case when a *ext*(read) is overlapped by a *ext*(write). If the corresponding *int*(write) is not performed (i.e., value is unchanged), then register x will just return the old value, and this *ext*(read) will be correct. If the corresponding *int*(write) is performed, x may return either 0 or 1. However, both 0 and 1 are in the feasible value set of this *ext*(read) because the overlapping *ext*(write) is toggling the value of the register. Therefore, the *ext*(read) will be correct. Figure 13.8 now reduces to Figure 13.10.

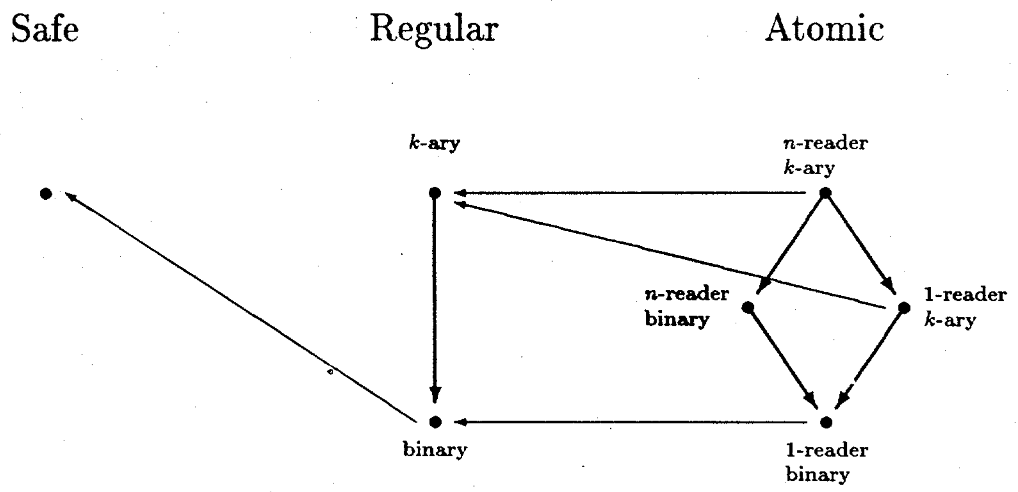


Figure 13.8: Collapsed Implementation Relationships

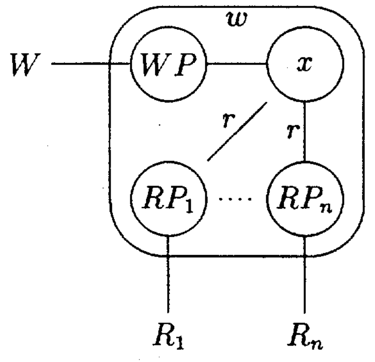


Figure 13.9: Binary Regular Register from Binary Safe Register

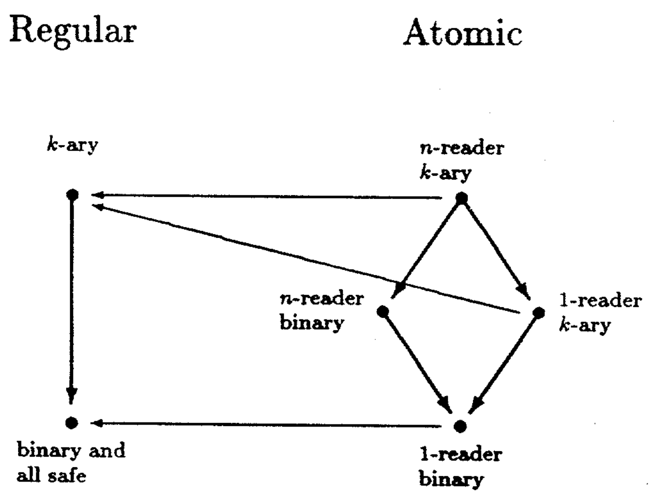


Figure 13.10: Collapsed Implementation Relationships

13.3.5 K-ary Regular Register from Binary Regular Register

We can implement a k-ary regular register using k binary regular registers as in Figure 13.11. 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 *ext*(write)(v)

int
(write) 1 to x_v. Then, in order, *int*(write) 0 to $x_{v-1}, \ldots, x_0$.

Code for *ext*(read)

int
(read) $x_0, x_1, \ldots, x_{k-1}$ in order until some $x_v = 1$ is found. *ext*(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 13.4 *If a ext(read) R sees a* 1 *in x_v, then v must have been written by a ext(write) which is feasible for R.*

Proof: Suppose not. Then R sees $x_v = 1$, and neither an overlapping or immediately preceding *ext*(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

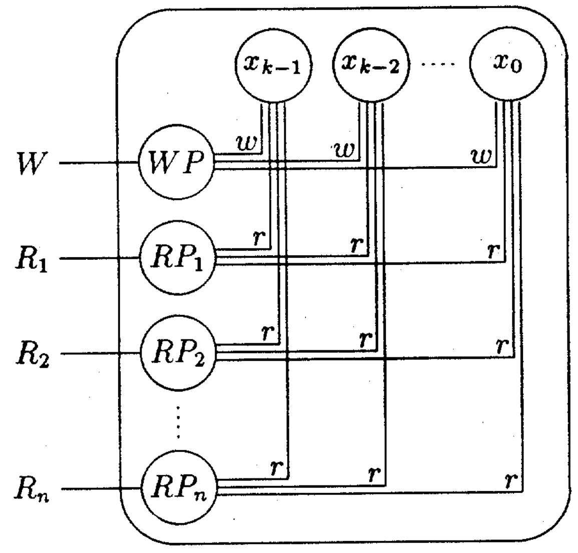


Figure 13.11: K-ary Regular Registers from Binary Regular Registers

W_1 which completely precedes R (otherwise W_1 would be a feasible *ext*(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 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 *int*(write)(1) to $x_{v'}$ completely follows W_1 and completely precedes the *int*(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 *int*(read) of $x_{v'}$ in R. Therefore, there exists some *int*(write)(0) to $x_{v'}$ which follows the *int*(write)(1) to $x_{v'}$ and either precedes or overlaps the *int*(read) of v' in R. But this can only happen if there is an intervening *int*(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 *int*(write)(1) to $x_{v''}$ completely precedes the *int*(read) of $x_{v''}$ in R because the *int*(write)(0) to $x_{v'}$ either precedes or overlaps the *int*(read) of $x_{v'}$ in R, and $v'' > v'$.

Note: The case when v is the initial value can be treated similarly. ■

The implementation relationships now collapse as shown in Figure 13.12.

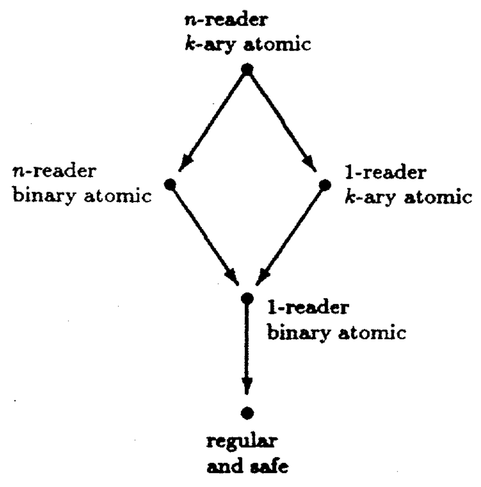


Figure 13.12: Collapsed Implementation Relationships

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 14: October 30

Lecturer: Nancy Lynch *Scribe: Shen-Wei Huang*[18]

14.1 Register Constructions: the End of the 1-writer case

The last time we had reduced the relationship diagram to Figure 14.1. Recall that we are dealing here with 1-writer registers. The case of multi writers will be treated in the next section.

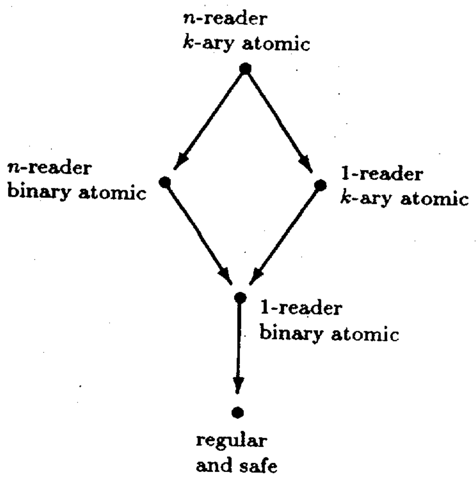


Figure 14.1: Gollapsed Implementation Relationships

14.1.1 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 14.2. 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 $\{\text{red}, \text{blue}\}$.

[18]Based on the complete lecture notes from 1988 scribed by Christopher Colby

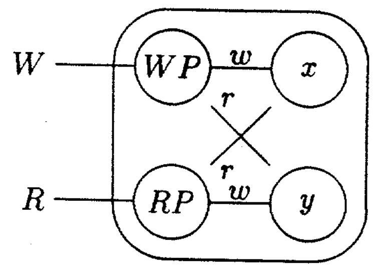


Figure 14.2: 1-Reader Atomic Register from Regular Registers

Code for *ext*(**write**)(v)

```
Remember value of $x$ locally as $x$.new.
newcolor $\leftarrow \neg(int(\text{read})\ y)$
oldvalue $\leftarrow x$.new
for $i = 1, 2, 3$ do $int$(write) $\langle$oldvalue, $v, i$, newcolor$\rangle$ to $x$.
```

Code for *ext*(**read**)

```
Remember last two reads in $x'$ and $x''$.
$x'' \leftarrow x'$
$x' \leftarrow int(\text{read})\ x$
$int$(write) $x$.color to $y$.
case
    num = 3:
        new-returned $\leftarrow$ true
        $ext$(return) $x'$.new
    $x'$.num < 3 and $x'$.num $\geq$ ($x''$.num $-$ 1) and new-returned and $x'$.color = $x''$.color:
        $ext$(return) $x'$.new
    else:
        new-returned $\leftarrow$ false
        $ext$(return) $x'$.old
end case
```

Read [Lamport86] for the correctness proof. The implementation relations now collapse as in Figure 14.3. This concludes Lamport's construction. At this point we still have a gap in between n-reader, 1-writer atomic registers and 1-reader, 1-writer atomic registers. It

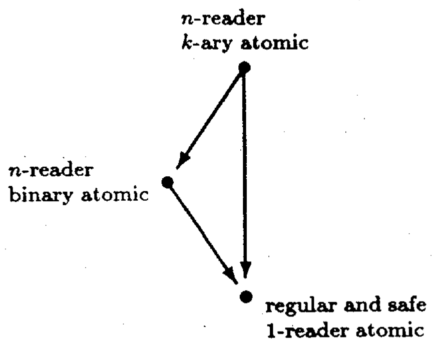


Figure 14.3: Collapsed Implementation Relationships

was closed in a couple of other papers: [BurnsP87], [NewmanW87,] and [SinghAG87]. But these algorithms are very complicated and time-consuming, and need more work to become practical algorithms.

14.2 Multi-writer Register

The implementation of n-writer registers with 1-writer registers has been tackled in many different papers. [Bloom87] treated the case of 2-writer registers, [VitanyiA86] and [Peterson-Burns-Schaffe] were buggy. We will discuss here Bloom's 2-writer algorithm, the unbounded case of Vintanyi-Awerburch's algorithm, Herlihy-Loui-Abu-Amara's impossibility result for constructing atomic test-and-set registers from atomic read-writer registers and Attiya's atomic snapshots of shared memory.

But first, let us give a remark about how atomic registers relate to our I/O model. There are two basic models of I/O that we will refer to as A and B. A is the model where we have an I/O automaton for each process and each variable. B is the model where the modeling of the whole system is done through a *single* big automaton. A expresses that the object responds to all accesses as if they happened at some particular time in the interval; B expresses that they happened truly indivisible. We would like to say that A "simulates" B in the sense that all the fair behaviors of A are also fair behaviors of B (considering only the external interface). But this statement needs to be proven carefully. The idea is to take a fair execution of A and to interchange the events so that each invoke-respond pair on physical registers occurs consecutively, then to replace the pair by a single step in system B. To do this interchange, we needs to know that consecutive steps do not affect each other, so that they can happen in either order. In order to do this, the following restrictions are required on A. These conditions have been worked out carefully in Ken Goldman's PhD thesis:

1. The system's interface with the outside world consists of pairs of invocation and re-

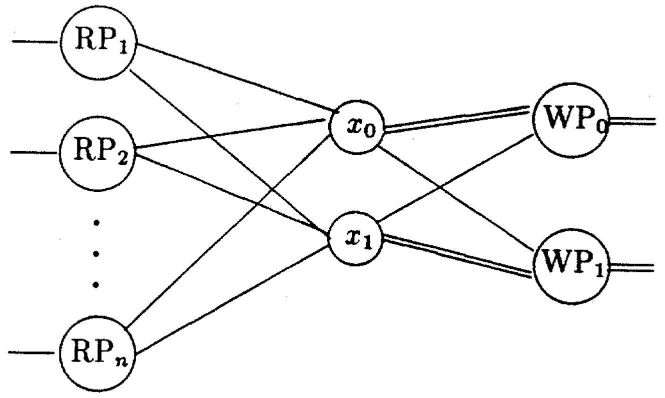


Figure 14.4: Architecture of Bloom's 2-writer atomic register construction. Lines between processes and registers denote access channels. Read access channels are shown as single lines and write access channels as double lines.

sponse,

2. The user invocations are sequential on each line,
3. The processes take steps accessing shared objects sequentially,
4. The processes only have locally controlled steps enabled in between their invocations and responses.

In this setting, Ken's results say that, if the processes only take one step at a time and cannot take steps except during the protocol, then, without loss of generality, we can think that the physical accesses are indivisible.

14.3 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. The algorithm is simple but does not apparently generalize.

The construction of the 2-writer atomic register is shown in Figure 14.4.

It consists of two atomic 1-writer $n+1$-reader registers (x_0 and x_1), n read processes (RP$_1 \ldots$ 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. When a writer writes, it tries to make sum of tags mod 2 as its own index.When a reader reads, it

Algorithm BL using binary tags

Shared variables: x_0, x_1

```
$x_0$ is initially $(v_{\text{init}}, 0)$
$x_1$ is initially $(v_{\text{init}}, 0)$
   where $v_{\text{init}}$ is the initial value of the logical register
```

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 $v_2$
```

Algorithm BLS using integer tags

Shared variables: x_0, x_1

```
$x_0$ is initially $(v_{\text{init}}, 1)$
$x_1$ is initially $(v_{\text{init}}, 0)$
   where $v_{\text{init}}$ is the initial value of the logical register
```

Algorithm for WP$_i$:
(writing a value v)

```
read $(v', u')$ from $x_{\neg i}$
$u \leftarrow (u' + 1)$
write $(v, u)$ to $x_i$
```

Algorithm for RP$_i$:

```
read $(v_0, u_0)$ from $x_0$
read $(v_1, u_1)$ from $x_1$
if $|u_0 - u_1| = 1$
   then $r \leftarrow i$ such
      that $u_i = \max(u_0, u_1)$
   else $r \leftarrow$ arbitrary
read $(v_2, u_2)$ from $x_r$
return $v_2$
```

Figure 14.5: Code of Bloom's 2-writer Construction

reads both registers and decide to choose one based on whether the sum of tags mod 2 is 1 or 0, it rereads the one it chose and return its value. The code appears in Figure 14.5.

14.3.1 Correctness

Bloom's paper has a very detailed and low-level proof. It starts with any well-formed fair behavior of this system, and shows that it is possible to insert dummy "perform" actions so that shrinking the invocations and responses to these perform actions gives a correct serial behavior for the R-W object. Unfortunately there is not much insight in this proof so that we will instead sketch another one that works at a higher level.

We will work implementing the possibilities-mapping idea. Like for ABP, we will give a version of the algorithm that uses a sequence of integer numbers instead of a sequence of bits. But here the reduction will not only be through the lower bits but through the *second* lowest-order bits.

reduction strategy: writer WP_0 sets the second lowest-order bits of the tags to the same value whereas WP_1 sets them to different values. For instance, assume that each of the two writers repeatedly sets its tag value to be the other one's + 1. Figure 14.6 illustrates this situation.

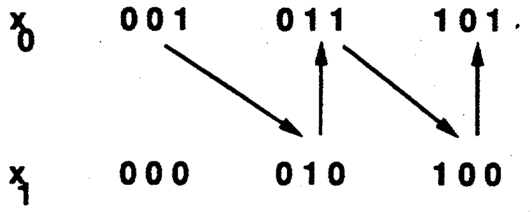


Figure 14.6: W_0 and W_1 increase their tag values

We see that W_0 acts as if trying to set equal the second order bits, whereas W_1 tries to set them different.

Lemma 14.1 *There is a possibilities mapping from Bloom bit algorithm to Bloom's integer sequence algorithm.*

Proof: (Sketch) The idea of the proof is to show that any step in BL can be emulated by a corresponding step in BLS.

When a writer sets its tag different from the other in BL, this can be emulated in BLS by setting its associated integer value to the other's plus one as indicated in the example of Figure 14.6.

When a reader sees t_0 and t_1 it decides to read some t_r. We show that the same register is allowed in BLS:

1. If the corresponding integer tag u_0 is one more than the corresponding u_1 then their second lower-order bits are equal. In this case, t_0 is equal to t_1, BL decides to read from x_0 and BLS also reads from x_0.

2. If the corresponding integer tag u_0 is one less than corresponding u_1 then their second lower-order bits are different; t_0 is different from t_1. Then both BL and BLS read from x_1.

3. For the remaining cases, BL reads some particular register; but in this case BLS allows to read *either*.

■

Hence we just have now to prove correctness for BLS. For this, it is useful to develop a sufficient condition for correctness involving partial orders. It will also be used to show the correctness of Vitanyi-Awerbuch's algorithm in the next lecture.

Definition regular sequence of a logical register

A sequence β of external actions of a logical register is *regular* if

- For each line, calls and responses alternate, starting with call,
- Each call has a corresponding later response.

Lemma 14.2 (Ordering Lemma) *Suppose a regular sequence β has a partial ordering $<$ of all operations (paired invocations and responses) satisfying:*

1. *If end_i precedes $begin_j$, then it cannot be the case that $operation_j < operation_i$.*
2. *$<$ orders all WRITE operations with respect to each other and orders all READ operations with respect to the WRITE operations.*
3. *The value returned by each READ operation is the value written by the last WRITE operation that is ordered before it in $<$. (Or the initial value, if no WRITE ever occurs before it.)*

Then β is a correct atomic register behavior.

Proof: We must show that we can augment β by adding in dummy actions $*_i$ between begin_i and end_i, and that each READ_i gets the value written by the WRITE_j such that $*_j$ must closely precedes $*_i$ (or the *initial* value if there is no preceding WRITE). The rule will be the following: we insert the $*$ for event i (denoted by $*_i$) immediately after the latest among $\{\text{begin}_i$, 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 $<$. ■

Claim 14.3 *Each $*_i$ is between begin$_i$ and end$_i$.*

Proof: 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 operation$_j$ $<$ operation$_i$. However, this contradicts the first condition on the partial order; so we conclude that $*_i$ is correctly between begin$_i$ and end$_i$. ■

Claim 14.4 *The precede-order on $*$ symbols is consistent with the $<$ order on operations: if operation$_j$ $<$ operation$_i$, then $*_j$ precedes $*_i$.*

Proof: Let operation$_j$ $<$ operation$_i$, then $*_j$ occurs after the last of begin$_j$ and all begin$_k$ where operation$_k$ $<$ operation$_j$, and $*_i$ occurs after last of begin$_i$ and all begin$_k$ where operation$_k$ $<$ operation$_i$. Thus, $*_i$ occurs after $*_j$ by definition. ■

Claim 14.5 *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)*

Proof: 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, 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 will prove that the algorithm correctly implements atomic register behavior. This will be the topic of our next lecture.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 15: November 1

Lecturer: Nancy Lynch *Scribe: S.J.Lee*

15.1 Bloom's 2-writer Construction - Continued

We now continue the discussion of Bloom's 2-writer algorithm using integer tags. To establish its correctness, it suffices to define a partial ordering on the operations and show that it has the properties described in the Ordering Lemma of Lecture 14. We will order the WRITE actions by their sequence number (if two get the same sequence number, they are by the same writer, so that we order them in the order that they occur). We will order each READ right after the WRITE whose value it gets and order the consecutive READs in the order which is consistent with the begin-end pair.

The integer tag algorithm clearly satisfies the properties 2 and 3 of the Ordering Lemma. It remains to show that it satisfies property 1 i.e., that this order is consistent with the begin-end order. We work by contradiction and assume that End_i precedes Begin_j whereas the operations are ordered in opposite order: $\text{Operation}_j < \text{Operation}_i$. There are four cases based on the nature of the operations i and j.

1. Suppose $\text{WRITE}_j < \text{WRITE}_i$.

   - seqno(i) = seqno(j). The same process originates the two operations and must order them consistently with begin-end order. But this is a contradiction with our hypothesis, so that we must be in the following case:
   - *seqno(i) > seqno(j)*. If the WRITEs were done by different writers, WRITE_j would see seqno(i) or greater, and would choose a sequence number greater than seqno(i), and this is a contradiction.

     If they are done by the same writer, then WRITE_i had to observe at least seqno(i) - 1 in j's register, so WRITE_j would also (observe seqno(i) - 1) and so would choose at least seqno(i), a contradiction.

2. Suppose $\text{READ}_j < \text{WRITE}_i$. This says that READ_j gets its value from some WRITE_k ordered before WRITE_i.

   - seqno(k) = seqno(i). This is impossible: WRITE_k precedes WRITE_i, and READ_j cannot get seqno(k) because it is overwritten before READ_j begins. Hence we must consider the following case:

- $seqno(k) < seqno(i)$. The value obtained by READ_j must come from the other writer (because if it was the same writer, it would be overwritten). The only value that could be around at end_i is the value of $\text{WRITE}_i - 1$, say u_i - 1. But if READ_j sees $u_i - 1$ (which it must do), it sees it the first time it reads the register. If it saw also u_i on first read, it would choose the other register. Hence it must see $u_i + 2$ or greater. But then *rereading* the $u_i - 1$ register would give a later sequence number (at least $u_i + 1$).

3. Suppose $\text{WRITE}_j < \text{READ}_i$ Our hypothesis means that READ_i sees either WRITE_j or some other WRITE ordered after it. But it can't see WRITE_j since it hasn't happened yet. Similarly, later WRITEs with the same sequence number occur after WRITE_j and READ_i can't see them. Therefore READ_i must see some WRITE_k with $seqno(k) > seqno(j)$. If WRITE_k were done by the same writer as WRITE_j, it begins after WRITE_j and READ_i could not see it, so it must be by the other writer. If WRITE_k actual contained write step occurs after begin_j, then READ_i can't see it, so that this actual write step must occur before begin_j. But then WRITE_j would see it (or would see a larger one) and would choose a larger sequence number: contradiction.

4. $\text{READ}_j < \text{READ}_i$ Our hypothesis means that READ_j gets the result of a WRITE_j that precedes the WRITE_i that READ_i sees (in sequence-number order). They cannot have the same sequence number, because the value would be overwritten before begin_j. Therefore the sequence numbers must be different. Let u_i denote the value in WRITE_i. As in case (2), the only value possibly around at end_i is $u_i - 1$. We conclude as in case (2) that READ_j couldn't read u_i - 1.

15.2 Vitanyi-Awerbuch's n-writer Construction

Vitanyi-Awerbuch [Vitanyi] developed a construction of a k-ary n-writer n-reader register from k-ary, 1-writer 1-reader registers, but the 1-writer registers must be unbounded in size. Figure 15.2 describes the construction and the algorithms of the read and write processes. This algorithm uses a $2n \times 2n$ matrix of registers that we can think laid out as in Figure 15.1.

The use of lexicographic order was already used in Lamport's algorithm. The second field of the tag is used as a tie-breaker.

Claim 15.1 *The Vitanyi-Awerbuch algorithm has an ordering among READ and WRITE operations that satisfies the conditions of the ordering Lemma.*

Proof: To prove our claim, we want to look back at the algorithm and order all significant operations in the way indicated by the Ordering Lemma: that is, we want to order all WRITEs with respect to each other, and all READs with respect to all WRITEs. The

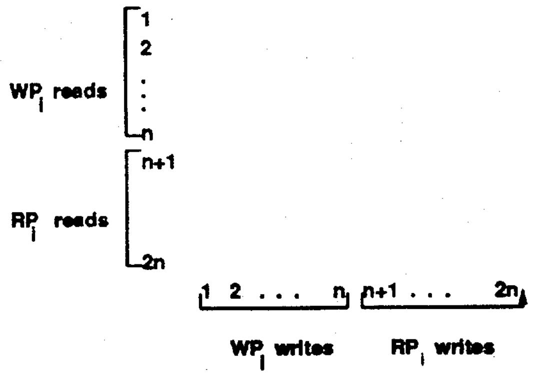


Figure 15.1: Matrix layout of variables in Vityani-Awerbuch's n-writer construction

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 $j \in \{1, \ldots, n\}$. The tags are ordered lexicographically.

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 15.2: The Vitanyi-Awerbuch n-writer register construction.

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 operation$_j$ $<$ 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 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 operation$_i$ $<$ 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.

∎

An interesting question is then to see how we could use the Ordering Lemma in the proof of some other more complicated multiwriter algorithms.

15.3 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. The essence of the proof is as follows: 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.) We will show in Section 15.3.2 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. We thus obtain the following theorem:

Theorem 15.2 *It is impossible to construct atomic test-and-set registers from atomic read-write registers.*

15.3.1 The Asynchronous Consensus Problem

Note that up to this point in the class, 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. Also we restrict failures to be simply stopping faults (so that Byzantine types of failures are disallowed) and assume a completely reliable message passing system.

We define formally the consensus problem as follows:

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.

Definition A *configuration* consists of the set of all process states together with the state of the message system.

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). Lemma 15.3 shows that the possibility of a fault precludes such a consensus protocol.

15.3.2 Proof of Theorem 15.2

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 15.3 *Every protocol that solves consensus in the presence of (at most) one stopping fault 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: $000\ldots0$, $000\ldots01$, $000\ldots011$, $\ldots$, $00111\ldots1$, $0111\ldots1$, $111\ldots1$. 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. ■

Lemma 15.4 *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 Lemma 15.3 tells us 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. ■

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 15.3.) Let π_0 and π_1 be steps of p and q, respectively.

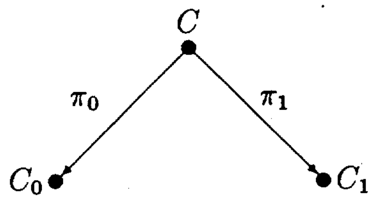


Figure 15.3: Deciding Configuration

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.

■

6.852 Distributed Algorithms	**Fall Semester, 1990**

Lecture 16: November 6

Lecturer: Nancy Lynch — *Scribe: Suwei Wu*

16.1 Atomic Snapshots

In this lecture, we will cover one last register result formulated by Afek, Attiya, et al involving wait-free algorithms. The algorithms are generally complex and, therefore, we will begin with some useful building blocks such as atomic snapshot objects. An atomic snapshot object is an entire memory divided into n words. It has n $update_i$ and n $scan_i$ lines. $Update_i(v)$ writes v into $word_i$ while $scan_i$ returns the vector of the latest values for each i. As usual , the atomic version of this object has the same responses as if shrunk to a point in the interval. Hence an atomic snapshot provides the "vision" of the whole memory at a given point.

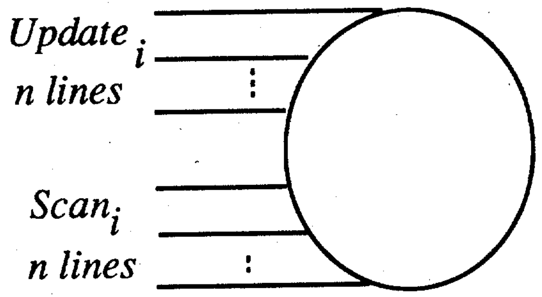


Figure 16.1: Atomic Snapshot Object

To implement a wait-free version of these objects, we use 1-writer, n-reader registers. These registers can be bounded or unbounded in size.

16.1.1 Unbounded Single-Writer Algorithm

The algorithm for unbounded registers is simple and is based on two observations.

Observation 16.1 *Suppose every update leaves a unique mark in its register. If 2 reads of all registers return identical values, where one read starts after the other completes, then the values returned constitute an atomic snapshot.*

The algorithm could have updates that just write v and the local sequence number to the register while the scanner keeps reading till it sees 2 identical vectors. However, this is not completely correct since the algorithm may never terminate. The solution in this case is discussed in the next observation.

Observation 16.2 *If a scan sees an updater change its value 3 times, then that updater executes a complete update operation within the interval of the scan.*

The illustration of the observation is shown in Figure 16.1.1 and an implementation of the algorithm follows.

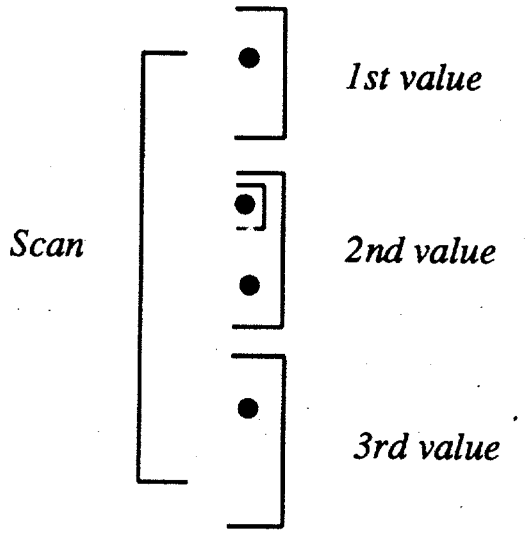


Figure 16.2: Observation 2 Illustration

Unbounded Single-Writer Algorithm
each *update$_i$* keeps a register with a

- value
- sequence number
- view (a vector of values)

Scan$_i$

- read all registers until we see 2 equal vectors or until one updater's component changes 3 times
- in the first case, return the repeated vector of values

- in the second case, return the view component associated with the 3rd version of the changed updater's register

Update$_i$

- do a scan as above - "embedded scan procedure"
- write the returned value, sequence number, and view of the scan

Proof: Assume the underlying reads and writes are done indivisibly. Then construct explicit points in the operations interval at which the read or write can said to have occurred.

Update We must serialize updates at a point where the write actually occurs. Thus, consider the sequence of writes that occurs in an execution α. After any prefix, there is a unique vector obtained by looking at the actual values in all the registers. These vectors, known as the "acceptable vectors", will be the only ones obtained by scans. In each scan, we need to pick a point in its interval to serialize it at. That point must be chosen so the vector returned by the scan is exactly the acceptable vector that exist after the sequence of writes that precede that point have been executed.

We must pick points for all the scans, real and embedded.

- First, consider scans, real and embedded that terminate with successful double collects. Pick any point between the end of its first collect and the beginning of its second. Then the vector returned is exactly the acceptable vector of register values at this point because there is no change in the designated interval.
- Second, consider scans, real or embedded that terminate with the default borrowed view. Consider these in the order of the completion of the scans. For each, note that the view it borrows is the result of another scan that is totally included within the given scan. So it has already gotten its point assigned. Choose the same point for these scans. The point is in the bigger scan interval since it is in the smaller.

By induction on the number of completions, we can show the value returned by each of these scans is exactly the acceptable vector at that point.

■

Time Complexity What is the time complexity of scan and update? Let us assume an upper bound of 1 on each of the procedure steps. In the no-failure case, the time complexity for each procedure is $O(n^2)$ because of the cost of each real and embedded scan. The scans might need to perform $O(n)$ collects as many as $2n$ times to get 3 changes from a register.

16.1.2 Bounded Single-Writer Algorithm

For this implementation, instead of keeping sequence numbers for *update*$_i$, keep n pairs of handshake bits for communication between *update*$_i$ and *scan*$_i$ for all i.

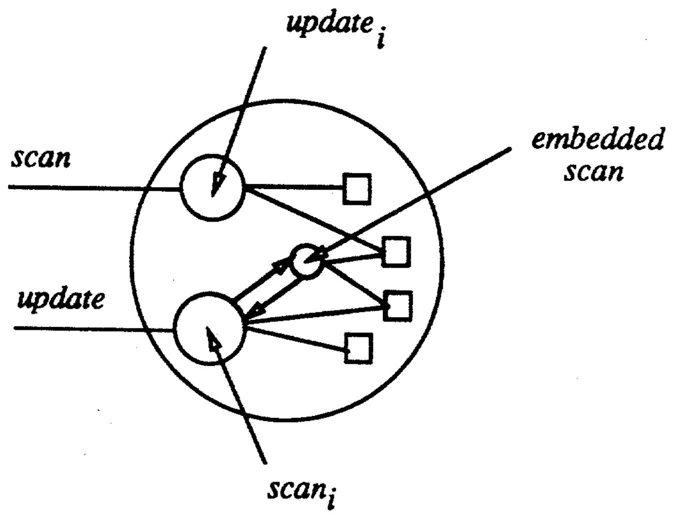


Figure 16.3: Bounded Register Implementation

The purpose of the sequence numbers was so that scanners could tell when *update*$_i$ has produced a new value. However, now the handshake bits will explicitly be used to communicate this.

The handshake bits are used as in the Peterson Fisher 2-process mutual exclusion algorithm. *Update*$_i$ sets the bits to be not equal while *scan*$_i$ sets the bits to be equal.

Specifically, for each (*update*$_i$, *scan*$_i$),

- $p_{i,j}$ in *reg*$_i$ is set to the negation of the value seen in $q_{i,j}$ when *update*$_i$ occurs
- $q_{i,j}$ in *reg*$_j$ is set during *scan*$_j$ to values read ¿from $p_{i,j}$

Also, *update*$_i$ has an additional toggle bit, *toggle*$_i$, that it flips during each write. This is to take care of a special case – to ensure that each write changes the register value. The complete algorithm can be found in page 10 of Handout 30.

Bounded Single-Writer Algorithm

Update$_i$

- first reads all the handshake bits

- then scans
- then writes negated handshake bits and negated toggle bit

Scan$_i$
for each tie *scan$_i$* tries to do a double collect,

- first it reads all the handshake bits and sets its own bits equal
- then it tries to perform the double collect where the tie is determined by the handshake bits, the toggle, and whether the handshake bit for $p_{i,j}$ is still equal to $q_{i,j}$
- otherwise, same as before

Proof: How do we prove this. The behavior of the algorithm does not seem close enough to use the mapping idea. However, we can try to mimic the previous proof on unbounded registers.

Again, assign points by the same rules as before – we need to know that (1) if a scan returns by the default case, then it returns a view resulting from another scan contained entirely within the first scan. Otherwise, (2) if the scan returns by successful double collect, the the vector returned is indeed the acceptable vector everywhere between the two collects.

The first condition is similar to before. After three scans with different views, we can choose the view that occurs after the first write and before the second write. The second condition, requires some original thought and is argued on page 11 of Handout 30. We must show that if a scan does successful double collects, then no write occurs between the end of the first and the beginning of the second – so the bits are sufficient to indicate changes.

We begin by contradiction. Suppose two reads by *scan$_i$* of r_i produce values of $p_{i,j}$ that are equal to $q_{i,j}$'s most recent values and toggle bit. Assume a write by i occurs in between the 2 reads. Consider the last such *write$_i$* – it must write the same handshake and toggle bit read by *scan$_j$*. Since during an update, *update$_i$* assigns to $p_{i,j}$ the negation of the value read in $q_{i,j}$, then the read must have preceded *scan$_j$*'s most recent write of $q_{i,j}$. So we must have this sequence

$read_i(q_{j,i} = \neg b)$	*update$_i$* reads handshake bit
$write_j(q_{j,i} = b)$	*scan$_j$* writes handshake bit
$read_j(p_{i,j} = b, toggle_i = t)$	first *scan$_j$* collect
$write_i(p_{i,j} = b, toggle_i = t)$	*update$_i$* write
$read_j(p_{i,j} = b, toggle_i = t)$	second *scan$_i$* collect

The *read$_i$* and *write$_i$* are part of the same *update$_i$* operation so then the 2 reads by *scan$_j$* return values written by the 2 successive *update$_i$* writes. The toggle bits, though, are identical – contradiction.

■

Atomic snapshots will be the last algorithm covered in the area of shared memory. The next topic is message passing which will begin with a cute, well-known, though not very practical algorithm.

16.2 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. The problem of finding an

MST reduces obviously to that of finding a spanning tree. 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.

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.

Connections between the MST Problem and Other Problems

As we just discussed rather informally, the MST problem 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.

In a graph, if that graph happens to be a tree, then breaking the symmetry caused by cycles is the hardest task of a general leader election algorithm.

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. To summarize all these problem relations, we can draw the following diagram:

$$\text{MST} \longleftrightarrow \text{ST} \longleftrightarrow \text{leader}.$$

In this theorem we implicitly consider channels that are not reliable and not FIFO. To prove it we assume there is such a protocol and look for a contradiction.

Assume we could produce a multiset T of packets, a finite execution α, and a k-extension $\alpha\beta$ such that:

- every packet in T is in transit from the t to r after the execution α (i.e.,T is a multiset of "old" packets).
- the multiset of packets received in β is a submultiset of T.

This last condition actually means that, for each *rec-pkt*$^{rt}(p)$ action happening in β, a *send-pkt*$^{rt}(p)$ had happened during α that had not been matched during α by a corresponding *rec-pkt*$^{rt}(p)$ action.

We could then derive a contradiction as wanted: Consider an alternative execution that begins similarly with α, but that does not have then any *send-msg* action occurring. All the packets in T cause the receiver to behave as in the k-extension $\alpha\beta$ and hence to generate an incorrect *rec-msg* action.

In other words, the receiver is confused by the presence of old packets in the channel, which were left in transit in the channel in α and are equivalent to those sent in β. At the end of the alternative execution, a message has been received without its being sent, and the algorithm fails.

In order to manufacture this situation, one further definition is necessary.

Definition 20.2.2 $T \underset{k}{<} T'$ if

- $T \subseteq T'$ (This inclusion is among *multisets* of packets.)
- $\exists$ packet p s.t. mult$(p,T) <$ mult$(p,T') \leq k$) (mult(p,T) denotes the multiplicity of p within the multiset T).

Lemma 20.3 *If α is valid, and T is a multiset of packets in transit after α has taken place, then either*

1. *$\exists$ k-extension $\alpha\beta$ such that the multiset of packets received by A^r in β is a submultiset of T, or*
2. *$\exists$ a valid execution $\alpha' = \alpha\beta$ such that*

    2.1. all packets received in β are sent in β,

    2.2. and $\exists$ a new multiset T' of packets in transit after α' such that $T \underset{k}{<} T'$.

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). More generally we could use the following basic strategy: start with all nodes as fragments and successively extend an arbitrary fragment with its MWOE, combining fragments where possible. This requires distinct weights since otherwise the procedure could create a cycle.

Earlier, we noted that in our version of the problem we will assume that all edge weights in our starting graph are actually distinct. The main property insured by the uniqueness of the edge weights is that every fragment has a unique MWOE. In this case, we have a second property that we can use to simplify our problem.

Property 16.4 *If all edges of a connected graph have distinct weights, then the MST is unique.*

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 16.2.1).

Besides the use of distinct edge weights, there are some other assumptions used in the Gallager-Humblet-Spira algorithm:

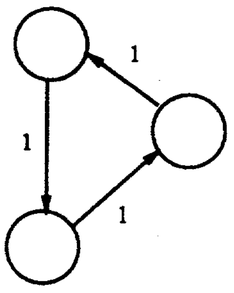


Figure 16.5: An unintended cycle is formed due to edges with equal weights.

- 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.)

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 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

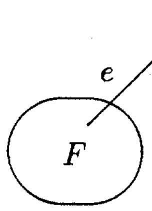


Figure 16.6: How does a node know whether an edge leads outside the fragment?

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 16.6.) 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 16.7.) In such a situation, the algorithm would require $\Omega(n^2)$ messages to be sent overall.

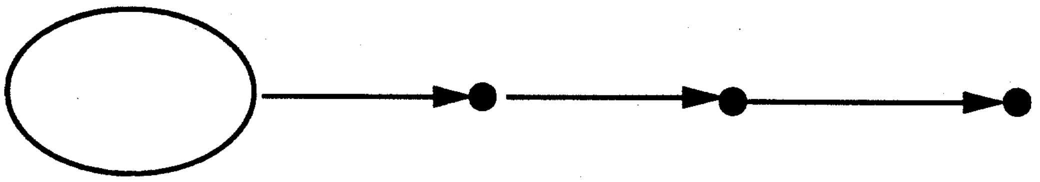


Figure 16.7: How do we avoid a big fragment growing by one node at a time?

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 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}.)

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 17: November 8

Lecturer: Nancy Lynch *Scribe: Philippe Park*[19]

17.1 Gallager-Humblet-Spira Minimum-Weight Spanning Tree Algorithm, cont.

In this lecture, we will continue the discussion of a distributed algorithm, due to Gallager, Humblet, and Spira [GallagerHS83], for computing the minimum-weight spanning tree (MST) of a graph. In Lecture 16, we introduced the concept of a minimum-weight outgoing edge (MWOE). It was shown that for any spanning forest in a graph (a collection of disjoint trees that include every vertex in the graph), that the minimum spanning tree must include the minimum-weight outgoing edge (MWOE). This principle forms the basis for well-known sequential minimum spanning tree (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). The Gallager-Humblet-Spira algorithm is also based on the concept of nodes that combine themselves into fragments of increasing size.

17.1.1 A High-Level Description of the Algorithm, cont.

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. However, this requirement is not very strong, since the algorithm could use processor IDs to break the symmetry, as we have seen in other algorithms in this course.

Recall that in lecture 16, we found that in order to reduce the message complexity from $O(n^2)$ to $O(n \log n)$, that it was necessary to implement a "balanced-tree algorithm" solution. Otherwise, the algorithm could proceed by having one large segment that picks up one node at a time. (See figure 17.1). This was achieved by associating *level numbers* with each fragment to keep fragments from being very unequal in size. We 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.

[19]Based on lecture notes from 1988 scribed by Mike Einsenberg

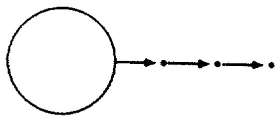


Figure 17.1: How do we avoid a big fragment growing by one node at a time?

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.

Let's 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'.

These two combining strategies are illustrated (in a rough way) by Figure 17.2. 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 17.2 is meant only to suggest the typical case.)

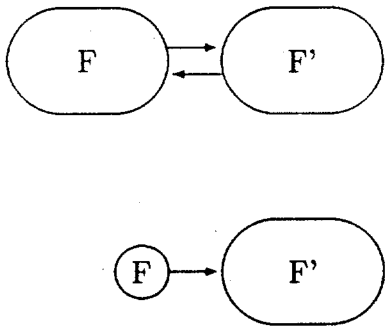


Figure 17.2: Two fragments combine by merging; a fragment absorbs itself into another

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.

It is interesting to note that fragments in the graph could each be modeled as one big giant I/O automaton, with the messages emanating from the core . The messages between fragments would be the external actions of these automata. We have observed in earlier lectures that I/O automata should always be input-enabled. In this particular algorithm, that means that if a fragment is "edge-enabled", that the I/O automaton should be written so that the processor connected to this edge gets a fair turn.

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.

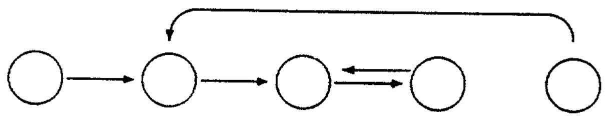


Figure 17.3: A collection of fragments and their minimum-weight outgoing edges.

Note that identifying a fragment by its core edge depends on the assumption that all edges have a unique identifier. If we continue to work with the assumption that the edges have a unique weight, then the weight could be the identifier.

We now want to show that this strategy, of having fragments merge together and absorb themselves into larger fragments, will in fact suffice to combine all fragments into a MST for the entire graph.

Claim 17.1 *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 17.3) 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

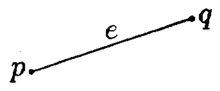


Figure 17.4: Node p wants to know if q is in the same fragment.

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 current fragment. Suppose we have a node p that "looks across" an edge e to a node q at the other end. (See Figure 17.4.) 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.

However, notice that we may have to reconsider the progress argument, since this extra precondition may cause progress to be blocked. But note, that we only need consider the set of fragments at the lowest level, since these always succeed in MWOE(p) calculations. If any of these fragments points to a higher level fragment, then it will become absorbed.

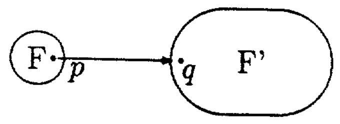


Figure 17.5: Fragment F absorbs itself into F' while F' is still searching for its own MWOE.

Otherwise, loops are formed and the calculations proceed as we have seen in earlier examples.

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 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 17.5.)

There are two cases to consider (consult Figure 17.5 for the labeling 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. One approach would be to use an I/O automaton for each fragment as discussed earlier and use possibilities mapping proof techniques. (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.

So each node finds the MWOE if it exists. All of this information is sent back to the nodes incident on the core edge via REPORT messages, who determine the MWOE for the entire fragment. A CHANGEROOT message is then sent back towards the MWOE, and the endpoint node sends a CONNECT message out over the MWOE.

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

$$\sum_{\textit{all fragments } F} \textit{number of nodes in } F$$

which is

$$\sum_{\textit{all level numbers } L} \left(\sum_{\textit{all fragments } F \textit{ of level } L} \textit{number of nodes in } F \right)$$

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. The level of complexity of this code seems to preclude invariant assertion proofs, at least at the detailed level of which messages are sent, local variables. Most of the discussion has been at the higher level of graphs, fragments, levels, MWOE's, etc. So, we should organize a proof which takes advantage of this high-level structure.

One promising approach is to apply invariant-assertion and other techniques to prove correctness for a high-level description of the algorithm and then prove independently that the code in fact correctly simulates the high-level description. (See [WelchLL88].)

A 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 possibilities mapping between the two automata.

There are still a number of open issues in this area. For example, there is no known upper bound for the time complexity of this algorithm.

17.2 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.*

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".

Now consider the resource allocation problem in the distributed networking environment. We can use the same external interface as in the shared memory model (described in more detail later) of "try", "enter critical region", "exit", and "enter remainder region" actions, but now, we use message-passing instead of shared variables to communicate. Since resource allocation requires mutual exclusion, we have the same requirements as in the shared-memory model.

A straightforward solution for implementing mutual exclusion 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

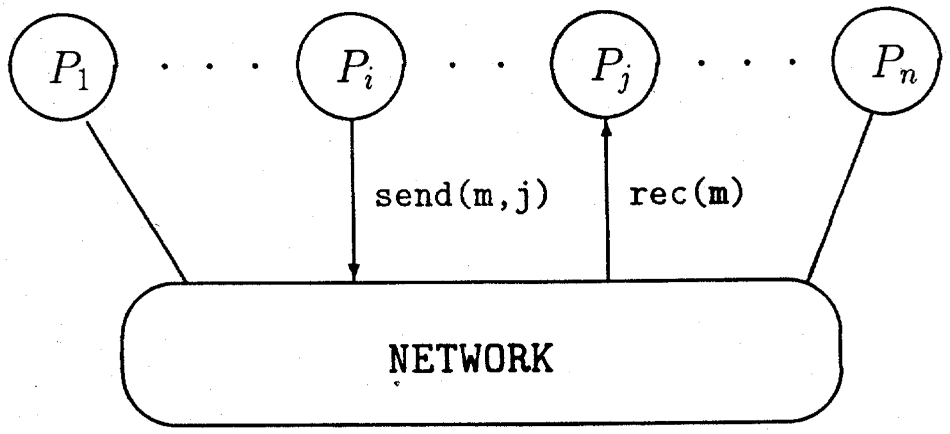


Figure 17.6: Distributed Network: processes communicating by message-passing.

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.

In the distributed networking environment, the requirement that processes are only allowed to take steps in the trying and exit regions, but not in the critical and remainder regions, is usually dropped for efficiency and resilience considerations. (In the shared-memory model, dropping this requirement would simplify the process considerably by using one process as arbiter, but this would also introduce a single point of failure.)

17.2.1 Modeling

In Figure 17.7, we model a node as two I/O Automata. The buffer (network) is also described as an I/O Automaton.

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.

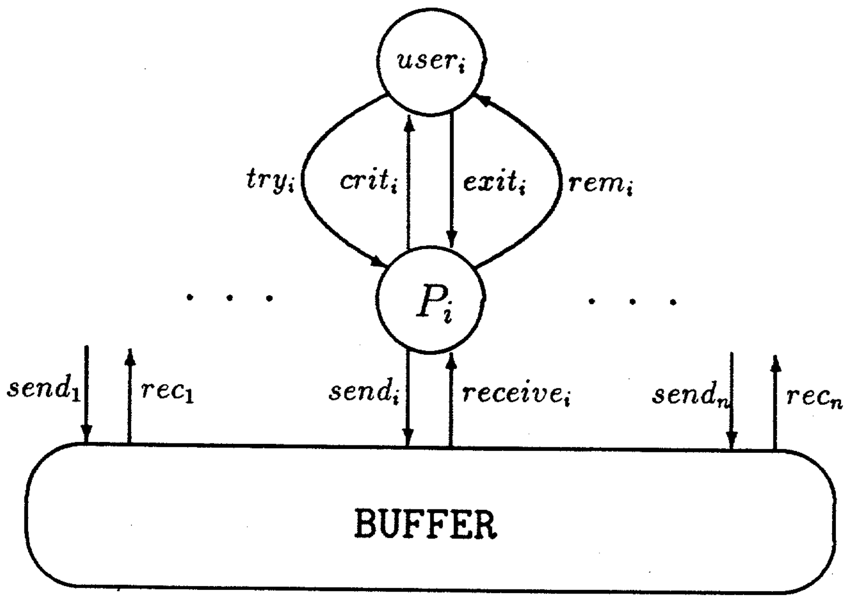


Figure 17.7: Distributed network modeled as I/O Automata.

17.2.2 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.
- *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, s = process steps. The worst-case time is $\approx (c + d + O(s)) \cdot n$. This time bound is bad because it has a $d \cdot n$ term, regardless of the load, and d may be big. However, analogous simulations in the shared-memory model seem much worse.

17.2.3 Lamport 1978

This paper: "Time, Clocks and the Ordering of Events in a Distributed System", is a famous one and worth reading. The problem of ordering events in a system can be viewed as that of implementing a distributed queue. The solution proposed in this paper is based on the concepts of timestamps and replicated database management techniques. 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*.) It is also important that each event in the system be *uniquely* identified, even as applied to unrelated events. One trick might be to append the site ID numbers in the low-order bits in order to break "ties'.

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

We assume that the network delivers messages between any pair of nodes in the same *ltime* order as they were sent. For example, in the space-time diagram of Figure 17.8, 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.

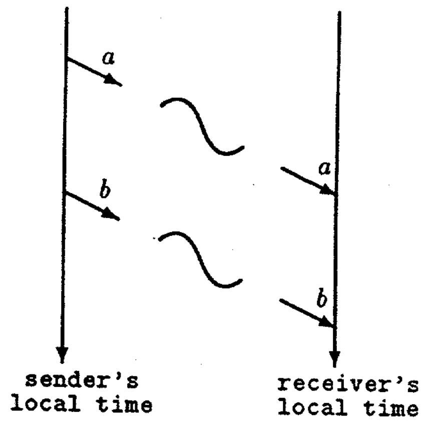


Figure 17.8: Space-Time diagram.

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. So while this algorithm appears to be centralized in nature, it is implemented in a distributed manner, and gains the associated benefits. 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 $\mathcal{C}$ 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 $2d + O(s)$, where d is the communication delay and s is local processing. We assume that the broadcast is done as one atomic step; if $n-1$ messages are treated separately, the processing costs are linear in n, but these costs are still presumed to be small compared to the communication delay d.

Recall that for the time complexity analysis, Lelann has the dn term.

17.2.4 Ricart & Agrawala 1981

This algorithm uses only $2(n-1)$ messages per request. It improves Lamport's algorithm (section 17.2.3) 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:

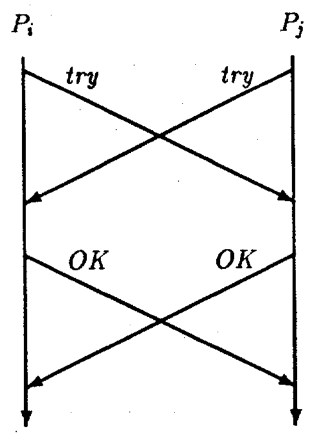


figure 17.9: both processes in C.

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 17.9, 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*.

17.2.5 Carvalho & Roucairol 1983

This algorithm improves on the previous one (Section 17.2.4) 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.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 18: November 13

Lecturer: Nancy A. Lynch *Scribe: K. Chiang*[20]

18.1 Resource Allocation in Networks

We already discussed the problem of resource allocation in Lecture 10. We considered there the case where processes communicated through shared memory. We will discuss in this lecture the case where communication among processes is done by *message passing* in networks and where each process needs a *conjunction* of resources to enter the critical section.

We keep the message-passing model of Figure 18.1 which we already employed for mutual exclusion.

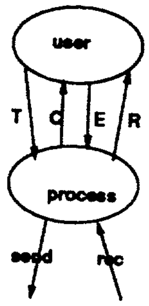


Figure 18.1: Messages exchanged

We will discuss quickly Lynch's solution to the Dining Philosophers Problem and then present Chandy-Misra's solution. This problem corresponds to the static case where the resource needed by each process are defined once and for all. The algorithm guarantees *mutual exclusion, progress* and *no-lockout.* We will then present Chandy-Misra's solution for the Drinking Philosophers Problem, which corresponds to the dynamic case in which a process may request a different set of resources every time it enters its trying region.

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.)

[20]Based on lecture notes from 1988 scribed by Rike Stille

18.2 Lynch's Approach

Hierarchical resource allocation may be used to solve the static problem, if arbiter processes are associated with each resource. These processes play similar roles as a centralized arbiter in mutual exclusion. User processes that wish to use a resource must communicate with the relevant arbiters in the appropriate order before securing the desired resource. They wait in queues when the resources are not immediately available.

The time complexity of the algorithm depends then only on such "local" factors as the chromatic number of the graph, the number of users per resource, and the message delay to the required arbiter processes.

18.3 Chandy-Misra dining philosophers

This algorithm guarantees exclusion, progress, and no lockout, as does the previous approach mentioned above. However, the time performance may be significantly worse. Recall that each process has a *fixed* set of resource requirements.

Description of the algorithm

The resources are referred to as forks, and the processes as philosophers. Each fork can be either clean or dirty. Dirty forks have been used, and must be cleaned before being used again. Cleaning only occurs when another philosopher requests the fork and the recipient of the request satisfies that request by sending it to the requester.

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 that you 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.

18.3.1 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 18.1 *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 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 categories, 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. ■

Notice that worst case execution time of this algorithm is potentially very bad, as the waiting chain, the height k mentioned in the above proof, can potentially span the entire network.

18.4 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.

18.4.1 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 18.2, where a dirty fork is located at the tail of each arrow.

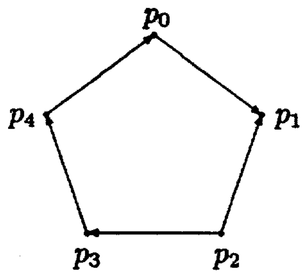


Figure 18.2: Acyclic graph H. All forks are dirty.

First, all philosophers enter wanting only their right forks. They all can get them: surely those with in-edges get them; but also p_2, since p_2 has its fork dirty and p_1 does not want it. So they all can use their right forks at once, dirtying them in the process. But this dirtiness orients the edges to the right, 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.

18.4.2 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 18.3.

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.

18.4.3 Drinking Philosophers Algorithm

While in R

- Satisfy all bottle requests.

- If the subroutine is in C, send it into E.

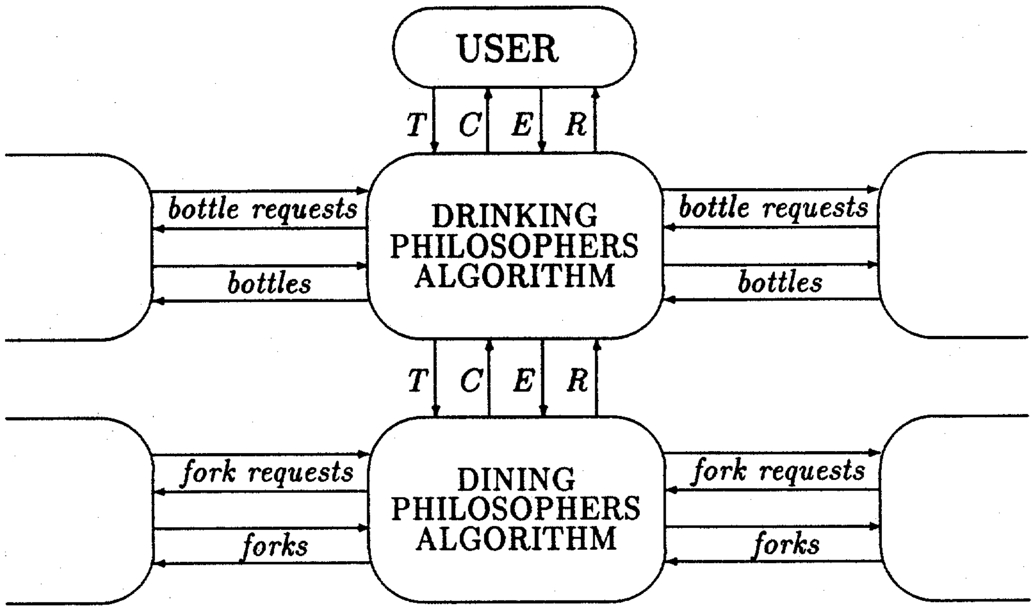


Figure 18.3: Proposed architecture for Lynch version of Chandy-Misra drinking philosophers algorithm.

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.

18.4.4 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 18.2 *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 18.3 *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 18.2. 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 18.4 *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 18.2, p_i reaches C. ∎

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 19: November 15

Lecturer: Nancy Lynch *Scribe: Jory Tsai*[21]

19.1 Consensus in Asynchronous Systems

The setting of this section is similar to the one of Lecture 15 where we showed that atomic read-write registers could not implement atomic test-and-set registers. We consider the general consensus problem in a completely asynchronous distributed system where processes communicate by **message passing**. No assumption is made about the relative speeds of the processes or about the length of any delays in message delivery. But, we assumed that the messages are eventually delivered by the message system. In future lectures, we will discuss algorithms dealing with the fault tolerant message system. We also 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). We begin assuming that all processes execute in a deterministic fashion—randomized solutions will be discussed in Section 19.2.

In this lecture we will show the surprising result that consensus is not possible in this setting. 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 a single process precludes any solution to the consensus problem.

19.1.1 The Consensus Problem

We restrict the consensus to be binary problem, $\{0, 1\}$, which is enough for us to prove the impossibility results. 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.

[21]Based on lecture notes from 1988 scribed by Andrew Sutherland

19.1.2 Modeling the System

We will use I/O automata to model the asynchronous system, as shown in Figure 19.1. (This presentation is somewhat simpler than the presentation in the original paper [FischerLP85].) Each process p_i, $1 \leq i \leq n$, is modeled 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 exactly one equivalence class in p_i's partition.[22]

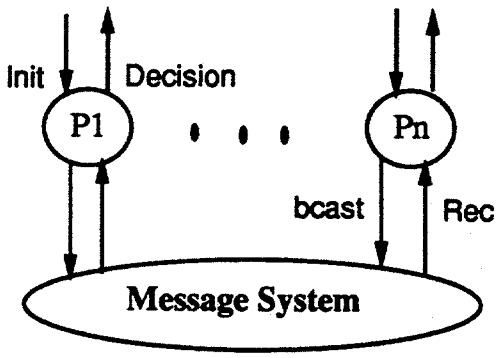


Figure 19.1: I/O Automata Model of the System

The message system is modeled 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 system 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

[22]If an algorithm makes use of countably many equivalence classes, then it may be simulated with a single class by dovetailing.

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 0solve 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 that a 1-RCP is necessarily also a 0-RCP.

19.1.3 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.

The following definition and lemma were already seen in Lecture 15.

Definition A finite execution α is *bivalent* if there exist extended configurations α_0 and α_1, both are reachable from α, such that in α_0 some process decides on the value 0 and in α_1 some process decides on the value 1. A finite execution is *univalent* if there is only one reachable decision value. A univalent finite execution is said to be *0-valent* if the reachable decision value is 0 and *1-valent* if it is 1.

Lemma 19.1 *Every 1-RCP has a bivalent initial prefix.*

We now present the main lemma, which claims that in the transition from a bivalent configuration to a univalent configuration there is always a single process which is responsible for the decision, and whose possible failure would prevent the whole 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 19.2 *Let A be any 0-RCP with a bivalent initial prefix. Then, A has a finite execution α and process p_i, such that:*

1. *α is bivalent*

2. *There exists a 0-valent extend finite execution α_0 of α, where the suffix has steps involving p_i* only.

3. *There exists a 1-valent extend finite execution α_1 of α, where the suffix has steps involving p_i* only.

The process p_i is called a decider *(See Figure 19.2).*

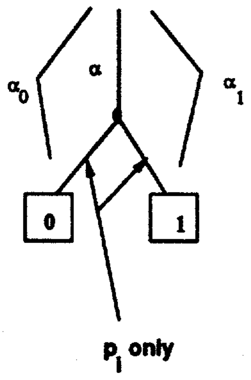


Figure 19.2: A decider

Proof: Suppose not. We start from a bivalent initial prefix α, and will construct a fair execution in which successive prefixes are all *bivalent*. This will violate the termination requirement for the consensus problem.

Consider a schedule consisting of a sequence of finite rounds, each round being organized in a round-robin fashion: in each round each process receives one step, and between two such process steps the oldest message pending (if any) is delivered. For convenience we will let *turns* denote process steps and message delivery steps. Delivering the oldest message ensures that every message that is sent is eventually received, so that the execution 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.

So it suffices to do a case analysis over the two different possible turns t (taking local steps or message delivery) and prove that we can extend the current execution with t so as to keep the execution bivalent.

Consider the case of a message delivery.

Consider the tree of all possible finite extensions of α in which m delivered just at the end. (See Figure 19.3.)

If any of these "leaves" is bivalent, we are done with this stage. So we consider the case where they are all univalent. Then we claim that there exists at least one 0-valent *and* at least one 1-valent among them: by symmetry it suffices to prove this fact for 0-valent. As

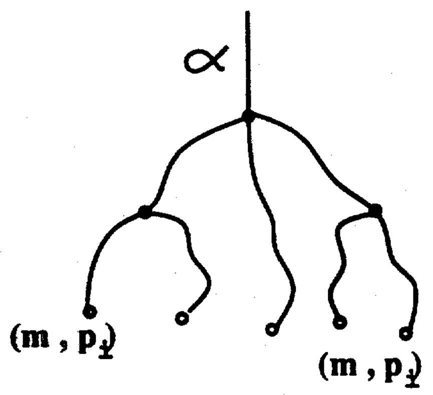


Figure 19.3: The tree of the extensions of α that deliver m at the end

α is bivalent there is an extension α_0 leading to a 0-decision. If (m, p_i) appears in the suffix of this extension then the state must be univalent right after this (m, p_i): by assumption it cannot be bivalent, and the extension α_0 is 0-valent. If (m, p_i) does not appear in the suffix of this extension then, by definition, m is still to be delivered, so that (m, p_i) is still enabled at the end of α_0 and hence can be appended to α_0. But α_0 is by assumption 0-valent. Hence the execution is of course still 0-valent after the addition of (m, p_i).

But as all the (m, p_i) leaves are univalent and as there exist a 0-valent and a 1-valent among the leaves of the tree, then somewhere in the tree there is an adjacent[23] pair of leaves such that one is a 0-valent (α_0) and the other is a 1-valent (α_1): see Figure 19.4 (a). (Recall that a node cannot have two edges labeled (m, p_i) departing from it: at a given point of the execution there is a ***unique*** oldest pending message. If it is not delivered during the current turn, a local process step is performed instead (corresponding to the edge "one-step" of Figure 19.4 (a)). But then for the next turn the same message (m, p_i) is still to be delivered.)

Call j the process taking the turn corresponding to the "one-step" of Figure 19.4 (a). If $i \neq j$ we can permute the actions "one-step" and (m, p_i) as indicated in Figure 19.4 (b): we get the same system configuration at the end. But one execution suffix is 0-valent and the other is 1-valent: contradiction!

On the other hand, if $j = i$ then p_i is a decider, and this contradicts our assumption that no decider exists.

[23]We say that two leaves are ***adjacent*** if their parents are connected by an edge in the tree.

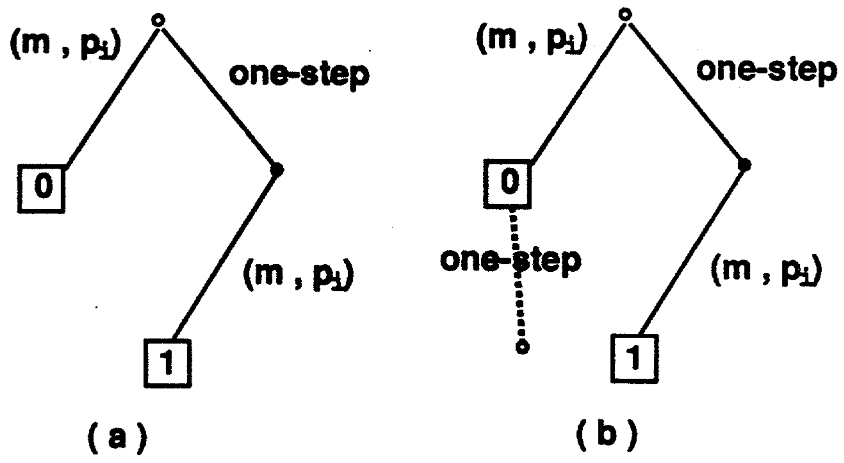


Figure 19.4: Two adjacent leaves (m, p_i) with different valence

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 19.3 *A 1-RCP does not exist.*

Proof:

Assume there exists a 1-RCP. Then by Lemma 19.1 it has a bivalent initial configuration. By Lemma 19.2 there must be an execution α at which point there is a decider p_i (cf. Figure 19.2.) Now consider an extension α_2 of α, 1-fair, in which p_i takes no further locally controlled steps. By the 1-RCP property some process p_j eventually decides; without loss of generality we can assume that it decides on the value 1. Remove from the suffix all steps involving delivery of messages to p_i, and get another suffix α'_2. As these steps do not affect the other processes these ones must still decide 1.(See Figure 19.5.)

But consider running α'_2 after α_0 instead of after α (See Figure 19.6). Then, since no step of p_i is involved in α'_2, they still decide 1, which is a contradiction to the 0-valence of α_0.

∎

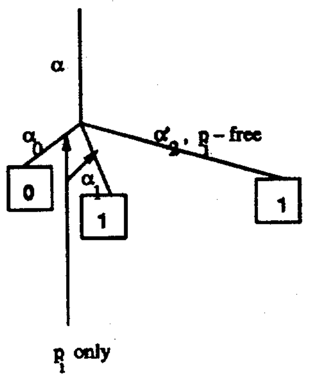


Figure 19.5: α_2' appended after α

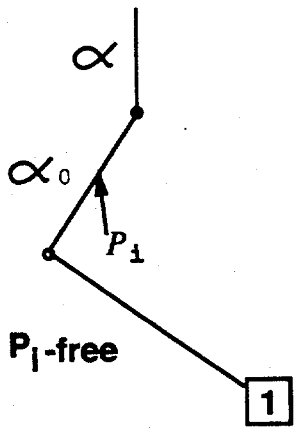


Figure 19.6: The final contradiction: a 0-valent execution α_0 with an extension deciding 1

19.2 Randomized Consensus Algorithm

In this section, we present Ben-Or's algorithm. We present in Section 19.2.1 the synchronous version of the algorithm. The asynchronous version is then presented in Section 19.2.3. The 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.

Process *p*'s code: (same for all processes)

```
repeat forever
Round 1:     1)   broadcast $first(x)$
             2-1) if $\geq$ $(n-f)$ messages. received. with value $v$
             2-2) then $x \leftarrow v$
             2-3) else $x \leftarrow nil$

Round 2:     3-1) broadcast $second(x)$
             3-2) if $\geq$ $(2f+1)$ msgs. recd. with value $v$
             3-3) then (DECIDE $v$; $x \leftarrow v$)
             3-4) else if $\geq$ $(f+1)$ msgs. recd. with value $v$
             3-5)      then $x \leftarrow v$
             3-6)      else $x \leftarrow random$
endrepeat
```

Figure 19.7: Ben-Or's Randomized Consensus Algorithm

19.2.1 Ben-Or's Randomized Algorithm

The code is shown in Figure19.7. 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 the executions are divided into *phases.* A phase consists of two rounds of the synchronous execution described in Figure 19.7. If $n \geq 3f+1$, then Ben-Or's algorithm satisfies 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.

We now establish the proof of correctness of the algorithm. This proof is inspired from a similar proof in [FeMi90].

For any non-faulty process i let $tally_{(i,r)}(v)$ denote the number of messages with value v received by i in round 2 of phase r.

Claim 19.4 *For all non-faulty processes i and j, for all pair of different messages v and w, for all phase r, if* $\text{tally}_{(i,r)}(v) \geq f+1$ *then* $\text{tally}_{(j,r)}(w) \leq f$.

Note that i and j are possibly equal.

*Proof:*Since $tally_{(i,r)}(v) \geq f+1$, at least one non-faulty player g must have broadcasted v in round 2 (line 3-1). Thus at least $n-f$ players sent v to g in round 1. Letting w ($w \leq t$) of these players be faulty, at least $n-f-w$ non-faulty processes send v to g in round 1. Therefore at most $(n-f)-(n-f-w) = w \leq f$ non-faulty processes can send any value other then v in round 1. In particular at most f non-faulty processes and hence $2f$ overall sent w to j in round 1. Since $2f < n - f$, no non-faulty process broadcasts w in round 2. Hence $tally_{(j,r)}(w) \leq f$ as we needed to show. ■

Agreement

If any nonfaulty process decides v in phase r of an execution, Claim 19.4 shows in particular that no one else can decide differently at phase r. Furthermore all other non-faulty processes must receive at least $(2f+1) - f = f+1$ messages v in round 2 and hence chose 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 $n - f$ messages with value b. Then, in Round 2 of the first phase, they will all broadcast b again and receive at least $n - f$ 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

The two previous properties jointly show that if a process i decides v during phase r then all other processes will have decided the same value by phase $r+1$. Consider then the case where no process i decides during round r. By Claim 19.4, for all processes i receiving more then f messages with some value v_i (during round 2 of phase r), this value v_i is actually independent of i: $v_i = v$. Then with probability $\geq 1/2^n$ all other processes will adopt the same value in line 3-6, ensuring the completion of the algorithm in phase $r+1$. Thus, the expected number of phases is at most 2^n.

19.2.2 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.

19.2.3 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. In fact, it can even tolerate strong type of byzantine fault, where processes can send any messages they like at any time.

The algorithm, shown in Figure 19.8, is also based on [Ben-Or83]. The algorithm assumes verifiable (but not necessarily secret) message channels. Although the algorithm needs additional processes ($n \geq 7f+1$), this number is reduced to $3f+1$ 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 $s(r, v)$, 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_i decides v at phase r. This can happen only if p gets $\geq (n-2f)$ occurrences of v, which by counting arguments guarantees that other nonfaulty processes get $\geq (n-4f)$ occurrences of v, hence they cannot decide on a different value at this phase. The reason is the faulty processes could lie and could leave out up to f nonfaulty processes allowed by p_i. Moreover, agreement will hold for the next phase if v is chosen at the end of the current phase by all other nonfaulty processes. This also shows that all nonfaulty processes terminate if any one does. We can argue that the probability of someone eventually terminates is 1.

Validity

If all start with same value, say 0, all nonfaulty processes decide in first phase on 0.

Termination

The agreement condition shows that if any process decides on v during a phase of the algorithm, termination will occur if all nonfaulty processes choose v at the end of the phase

Process p's code, for each phase r do:

```
Initially each process' initial value = $x$
 repeat forever
 Round 1:      broadcast $first(r, x)$
               wait for $(n - f)$ msgs. with value $first(r, *)$
               if $\geq (n - 2f)$ msgs. received with value $v$
               then $x \leftarrow v$
               else $x \leftarrow nil$

 Round 2:      broadcast $second(r, x)$
               wait for $(n - f)$ msgs. with value $second(r, *)$
               Let $v$ = value occurring most often, with $m$ = # of occurrences
               if $m \geq (n - 2f)$
               then (DECIDE $v$; $x \leftarrow v$)
               else if $m \geq (n - 4f)$
                    then $x \leftarrow v$
                    else $x \leftarrow random$
 endrepeat
```

Figure 19.8: Randomized Consensus Algorithm for Asynchronous Network

r. This implies that these processes will eventually choose v at the next phase. This event will occur with probability at least $\frac{1}{2^n}$, which value is very small. 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.

Consider the first nonfaulty p_i process reaches phase r of the algorithm, at the point where it receives at $(n-f)$ messages of type $first(r,*)$ messages. Let v denotes the majority value in the set of messages received (if a tie break, value will be arbitrary). We then make the following claims.

Claim 19.5 *At phase r, any $second(r,*)$ messages sent by a nonfaulty process must have value v.*

Proof: If some other value w was in $second(r,w)$ by nonfaulty process p_j, then p_j sees $\geq (n-2f)$ messages with value w in round 1. Then, by counting arguments, at least $(n-4f)$ messages with value w appear in the set of $(n-f)$ messages above. Since $n \geq 7f+1$, it is clear that $(n-4f)$ is still a majority of nonfaulty processes and thus $w = v$. ∎

Claim 19.6 *At phase r, the only value that can be forced upon anyone nonfaulty as its choice is v.*

Proof: To force a nonfaulty process to choose w, the process must receive at least $(n-4f)$ $second(r,w)$ messages. At least one of these processes is nonfaulty, so the value suggested is v. Consequently, the only 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 at the next phase.

Cryptographic assumptions can be used to cut down the number of rounds and improve the results. (See [FeMi90].)

19.3 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 distributed 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*, (*ltime*). 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 can 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 t. 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 t, 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. There is a well-formed notion of what's happened.

In the snapshot, we need to include exactly the information about:

- states of nodes after the local events up through time t;
- states of channels, which includes those messages have been sent but not received before time t.

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 19.9). 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.

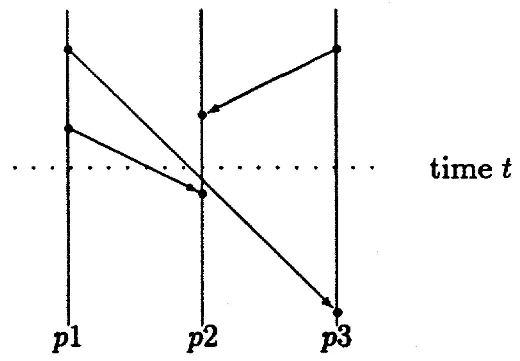


Figure 19.9: An execution represented by individual process timelines.

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? There are two most common types of applications:

- **Maintaining Database**: We might like to get a consistent state of a distributed database. For example, a bank audit over multiple branches of a bank.
- **Stable Property Detection**: The aim is here to detect some properties that *persist.* The two most common applications are:
  1. *Deadlock detection*: The problem is here to find out if every node is blocked, waiting for a result from some other node in order to proceed.
  2. *Termination Detection*: The problem is here 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.

19.3.1 Architecture of Global Snapshots

We describe here briefly the architecture we have in mind to establish snapshots and we present some problems that are associated with it. $\text{Algm}_1, \ldots, \text{Algm}_n$ correspond to the basic underlying algorithms. Each p_i is to snapshot the state of Algm_i and the state of all the links incoming to i. A problem is that p_i has no way to actually snapshot the state of Algm_i. (Action-based communication does not allow a composed process to see internal information as the state.) We need some way to modify this architecture in order to allow closer integration between p_i and Algm_i.

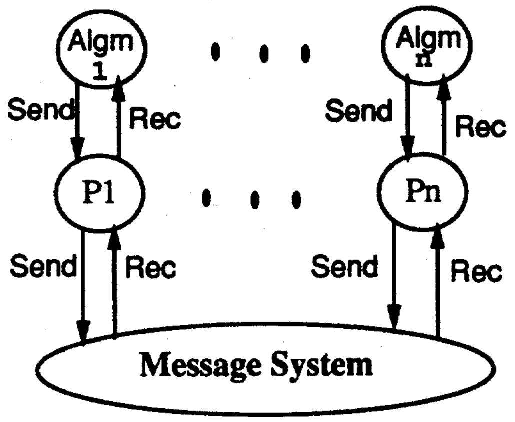


Figure 19.10: Architecture for Snapshot Interface

A first idea is then to assume that Algm_i has a special *snap* input action (and a corresponding response output) that tells the process to return its state at the time of the snap input action. (A model technicality is that it returns the value at some point later. We won't deal here with this problem.)

6.852 Distributed Algorithms	**Fall Semester, 1990**
Lecture 20: November 20	
Lecturer: Nancy A. Lynch	*Scribe: K. Chiang*

20.1 Chandry-Lamport Global Snapshots

A global snapshot is a picture of the state of every node, as well as the state of the channels which connect those nodes, the state of a channel being defined as the sequence of messages which have been sent out on it, but have yet to be delivered.

This algorithm guarantees that any global snapshot taken is *consistent* and *recent*; in other words, it accurately reflects the state of the system at some previous "recent" point in time.

To use this algorithm, the architecture presented at the end of previous class is assumed. Each node has two parts, Algm_i, a high level process that performs the desired computation at that node, and an underlying process p_i, which determines when to pass messages on, what new messages need to be sent, and when a snapshot should be taken.

20.1.1 The One Dollar Bank

Consider a bank with two branches p_i and p_j and total assets of a single dollar. We will let C denote the channel from p_i to p_j and D the channel from p_j to p_i. Suppose an audit of the bank is desired. Querying the nodes and channels at random will not work as desired, since their states could change between queries, adversely affecting the states of other components.

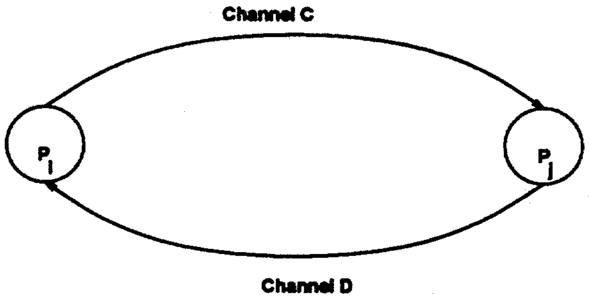


Figure 20.1: Two processes in communication

For instance, consider an execution where the p_i has the dollar initially. It then sends the dollar to p_j, which receives it at a later time. Even this simple execution may be recorded incorrectly. If the state of p_i is recorded prior to p_i's sending of the dollar, and then the states of C, p_j, and D are recorded, the dollar could appear in both p_i and C.

To circumvent the problem of duplication or loss of messages in a channel C connecting processes p_i and p_j, we introduce markers that delimit precisely the train of messages flowing in C between the time p_i and p_j do a snap. This leads to the following algorithm:

20.1.2 Algorithm

Assuming a channel C connects process p_i to another process p_j:

Rule 1: After p_i snaps its own state and before it sends any other message along C, it sends a special marker, denoted by the symbol #, on C.

Rule 2: Upon p_j's receipt of a marker,

- if p_j has already recorded its state, it records the state of C as the sequence of messages received along C after p_j took its own snapshot and before p_j received the marker.
- if p_j has not recorded its state, it does so now and records the empty sequence as the state of C.

Note that this algorithm can start asynchronously in one or more places. A process initiates the snapshot by recording its own state and propagating markers outward. Two or more processes that initiate the snapshot simultaneously will not conflict in the sense that the snapshot produced will belong to the set of valid snapshots. (We will the notion of valid precise in Theorem 20.1.)

The algorithm terminates when a marker has been received along all channels so that the states of all processes and of all channels have been snapped. This must happen sooner or later if the network is strongly connected so that the algorithm indeed terminates. (Recall that we are dealing with completely reliable channels.)

In the context of our architecture, the underlying processes may be expressed as I/O automata, using queues to keep track of incoming and outgoing messages and markers. A snap action can be performed by a process at any point before any receive-marker action is performed: this involves snapping the state and placing a marker at the end of all outgoing queues. At this point the process must start remembering (i.e.,placing in incoming queue) all incoming messages in *all* incoming queues. This snap action does not need being done instantaneously: it can allow processing of old things in the incoming queues. This must be finished though prior to a receive-marker action. If the queues are not finished being processed by the time such an action happens, then the snap is actually triggered by the reception of the marker and the state of the incoming channels is empty.

Call s_0 the initial state of the execution. We will let $\tau_\alpha(s_0)$ the state reached from s_0 by execution α.

This algorithm produces good snapshots in the following sense:

Theorem 20.1 *Let $\alpha = \alpha_1\alpha_2\alpha_3$ be the actual execution of the algorithm, where α_1 and α_3 exactly precede the start and follow the end of the snapshot algorithm. Let $s_1 = \tau_{\alpha_1}(s_0)$ and $s_2 = \tau_{\alpha_2}(s_1)$ the states reached after α_1 and $\alpha_1\alpha_2$ respectively. Then there is an execution $\alpha' = \alpha_1\alpha_2'\alpha_3$ such that $s_2 = \tau_{\alpha_2'}(s_1)$ and such that the snapped state s^* is reached from s_1 and such that s_2 is reached from it. Furthermore α_2' is a permutation of α_2 corresponding to a time reordering of unrelated actions.*

Proof:(Sketch) An action e in α is called a ***pre-recording*** event iff e is an action taken by a process p and p records its state *after* e in α. Similarly an action e in α is called a *post-recording* event iff e is an action taken by a process p and p records its state *before* e in α. An action is always pre or post recording. The key remark is that if a post-recording action e_{i-1} happens before a pre-recording action e_i then the two actions can be permuted and still lead to a valid execution: these two actions have to happen on different processes p_{i-1} and p_i and no message can be exchanged from p_{i-1} to p_i between these two actions. (If such a message was exchanged, p_i would receive a marker and e_i could not be a pre-recording event!) By induction all actions can then be reordered so that all pre-recording events happen before any post-recording event. The state s^* of the theorem is then the global state after all pre-recording events. ■

The preceding proof is based on the fact that, in the *global* system, an action e_i can be delayed past some other action e_j without modifying the *local* views of the execution held by each single process, as long as the process taking action e_j is not informed of e_i having happened. Let p_i and p_j denote processes taking these two actions. In more formal terms the preceding means that there is no chain of messages originated from p_i after the point at which e_i is performed and arriving at p_j before e_j is performed.

In terms of Lamport's partial ordering, the events at two different nodes can be reordered with respect to each other, preserving the partial orders within the nodes themselves. For instance the actual execution could be the one of Figure 20.2 whereas from the point of view of the snapshot it is as if the execution of Figure 20.3 had happened.

20.1.3 The Two Dollar Bank

To see this algorithm in action, consider the same two branch bank given above. This time, however, there are two dollars in the bank. Initially, each branch has exactly one dollar (See Figure 20.5). Each process can send as many dollars as it has.

We will illustrate the execution with diagrams. The first one, Figure 20.4 is an example explaining the symbols used.

Consider the following execution of the algorithm:

1. p_i records its state as having a dollar, sends a marker onto C, and then sends its dollar onto C (See Figure 20.6).

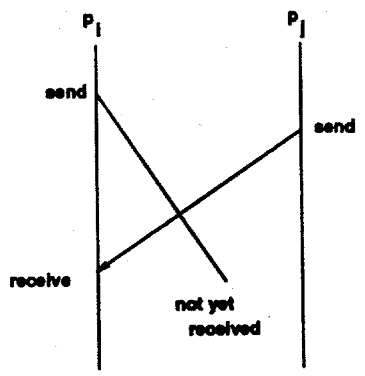


Figure 20.2: Some execution ...

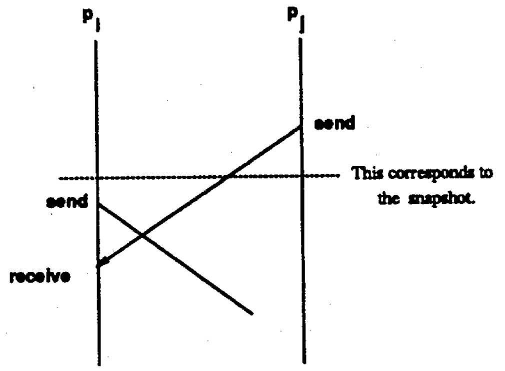


Figure 20.3: ... and another one corresponding to the snapshot.

2. p_j sends its dollar and p_i receives it (See Figure 20.7).
3. p_j receives the marker on C, so it records its state and that of channel C as empty (See Figure 20.8).
4. p_j sends a marker onto D (See Figure 20.9).
5. p_i receives the marker on D and records the state of D as having a dollar in it (See Figure 20.10).

Note that the state recorded by the snapshot has one of the dollars in p_i and the other in the channel D from p_j to p_i. This state did *not* occur in the actual sequence. Nevertheless it is reachable from the initial state (we can reach it by having p_j send a dollar on D), and from it the final state is reachable (by having p_i send its dollar on C and the dollar in D being delivered).

This is the reachability property expressed by theorem 20.1.

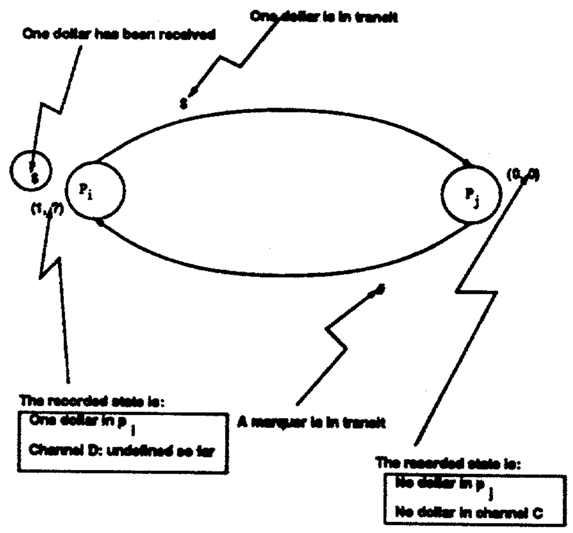


Figure 20.4: Explanation of the notations used

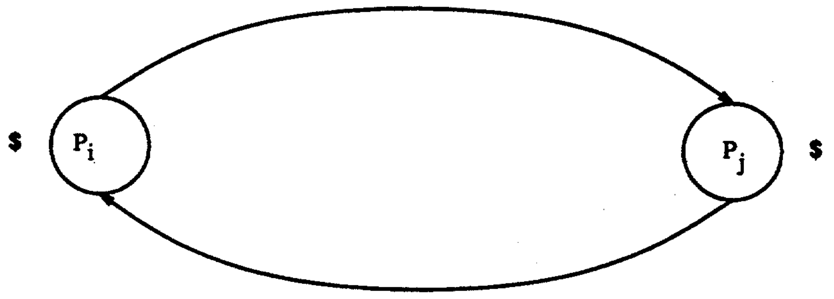


Figure 20.5: Initial Configuration

20.1.4 Stable Property Detection

A property is considered *stable* if, once it is true in a given state, it remains true for all states reachable from that given state. Thus, if a property P holds at some point of an execution, it continues to remain true until the end of this execution. Equivalently, if P is not true at a point in a sequence, then it was never true at any previous point in that same sequence.

The algorithm can be used to detect termination. If there is no external input, a system is quiescent if no actions are enabled and no messages are in transit. This is a stable property and can be determined by some central process examining a global snapshot, although convergecast along a spanning tree could be used instead in order to optimize the communication involved in the snapshot: instead of actually sending the whole state to one process, everyone can just check locally and fan in a bit saying that the state has been recorded in the subtree.

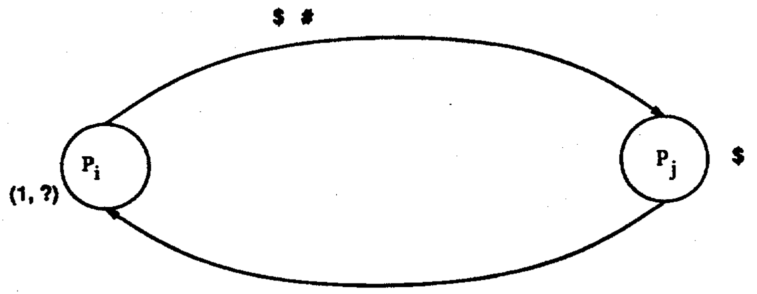


Figure 20.6: p_i sends a Marker and its Dollar

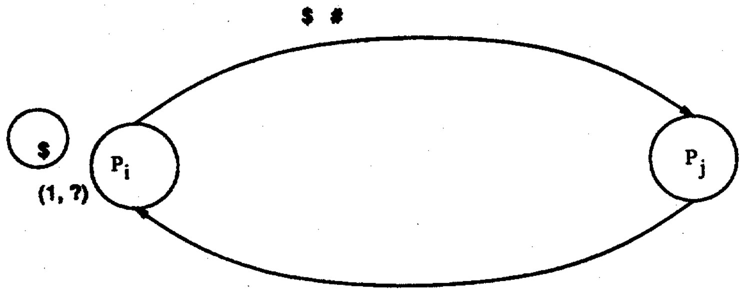


Figure 20.7: p_j sends a Dollar, p_i receives it

The algorithm can also be used for deadlock detection in a distributed graph. Deadlock is characterized by the fact that all processes are waiting for other processes to do something. Deadlock amounts to a waiting cycle. Hence deadlock detection can be achieved by taking a snapshot, sending all the information to some process and then doing an ordinary centralized cycle detection algorithm.

20.2 Datalink Protocol Impossibility Result

In the beginning of this course we studied the alternating bit protocol, designed to deliver messages reliably from a transmitter to a receiver in the order they were sent. We recall the set-up in Figure 20.11. The channels C^{tr} and C^{rt} connecting the transmitter and the receiver cannot duplicate messages but can can lose them. However, if infinitely many messages are sent, infinitely many messages are required to be delivered (This is a liveness condition). In addition, the channels are FIFO.

The user actions are *send-msg*(m) and *rec-msg*(m) for $m \in M$. The other actions involve sending and receiving of packets and are internal actions of the datalink protocol.

If the restriction on FIFO channels is removed, the alternating bit protocol with sequence

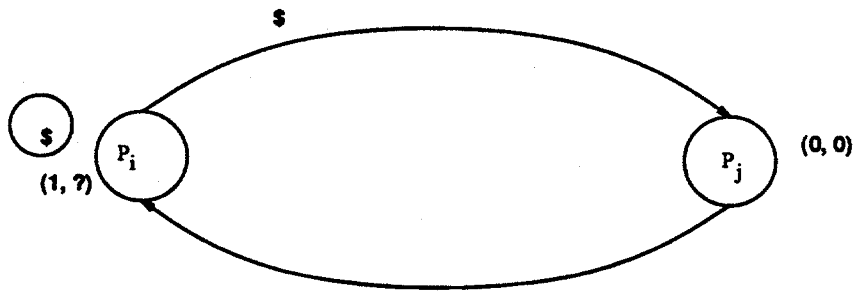


Figure 20.8: p_j receives the Marker

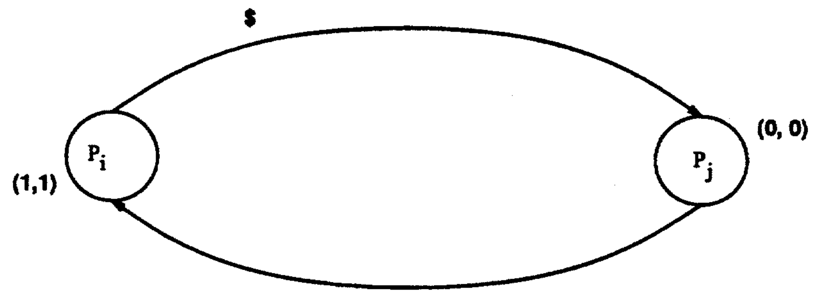


Figure 20.9: p_j sends a Marker

numbers still holds. However, the bit-only version of the protocol fails, since if more than two messages are sent, messages cannot be distinguished.

The bit-only version is one instance of a bounded header protocol. In general, such a protocol has messages of the form (m, h), where $m \in M$ and $h \in H$. The message alphabet $|M|$ is finite and the header alphabet $|H|$ is also finite.

The failure of the single-bit protocol suggests that no protocol with bounded size headers will function correctly, as recent work [FeLyMa90] by Fekete, Lynch, and Mansour shows.

In order to prove this impossibility result, a technical restriction on the number of packets used to send any message m is required.

A finite execution is *valid* if the number of *send-msg* actions is equal to the number of *rec-msg* actions. (This means that the datalink protocol succeeded in sending all the messages m provided by the user i.e.,all the messages m for which an action *send-msg*(m) occurred.)

the following definition expresses that in order to thus successfully deliver any message the datalink protocol only needs to send a *bounded* number of packets over the channels.

Definition 20.2.1 (k-boundedness) If α is a valid execution, an extension $\alpha\beta$ is a k-extension if:

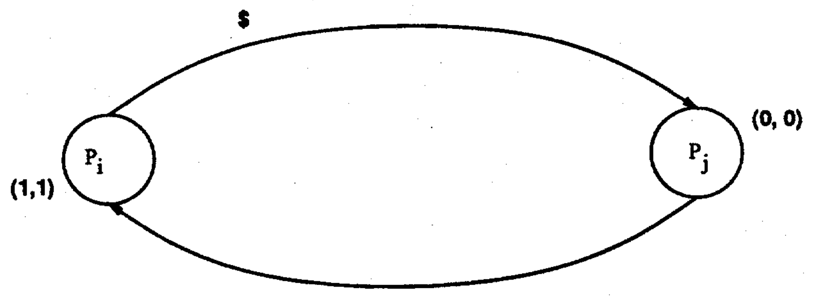


Figure 20.10: p_i receives the Marker

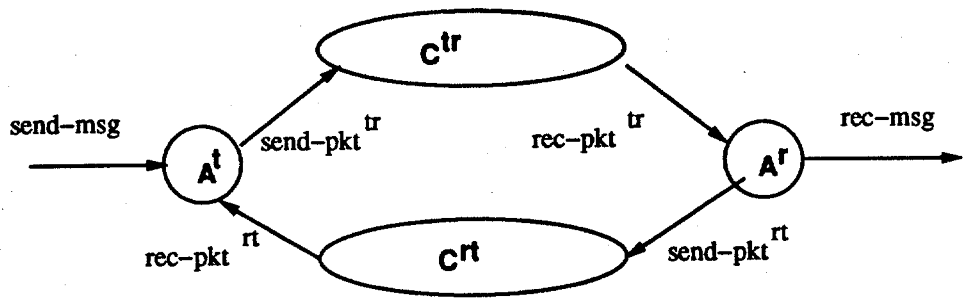


Figure 20.11: The ABP set-up

1. In β, the user actions are exactly the two actions *send-msg*(m) and *rec-msg*(m) for some given message m. (This means that exactly one message has been sent successfully by the protocol.)
2. All packets received in β are sent in β (i.e.,no old packets are received).
3. The number of *rec-pkt*^{tr} actions in β is less than or equal to k.

A protocol is *k-bounded* if there is a k-extension of α for every message m and every valid execution α.

Remark: Part 2 of the previous definition does not mean that all packets received in β are "physically" sent in β. What it means is that, for all $p \in P$, if a *rec-pkt*$^{tr}(p)$ action occurs in β then some corresponding *send-pkt*$^{tr}(p)$ action must have happened also in β. (The same condition holds with *rec-pkt*$^{rt}(p)$ and *send-pkt*$^{rt}(p)$ actions.)

20.2.1 Impossibility Result

Theorem 20.2 *There is no k-bounded datalink protocol.*

In this theorem we implicitly consider channels that are not reliable and not FIFO. To prove it we assume there is such a protocol and look for a contradiction.

Assume we could produce a multiset T of packets, a finite execution α, and a k-extension $\alpha\beta$ such that:

- every packet in T is in transit from the t to r after the execution α (i.e.,T is a multiset of "old" packets).
- the multiset of packets received in β is a submultiset of T.

This last condition actually means that, for each *rec-pkt*$^{rt}(p)$ action happening in β, a *send-pkt*$^{rt}(p)$ had happened during α that had not been matched during α by a corresponding *rec-pkt*$^{rt}(p)$ action.

We could then derive a contradiction as wanted: Consider an alternative execution that begins similarly with α, but that does not have then any *send-msg* action occurring. All the packets in T cause the receiver to behave as in the k-extension $\alpha\beta$ and hence to generate an incorrect *rec-msg* action.

In other words, the receiver is confused by the presence of old packets in the channel, which were left in transit in the channel in α and are equivalent to those sent in β. At the end of the alternative execution, a message has been received without its being sent, and the algorithm fails.

In order to manufacture this situation, one further definition is necessary.

Definition 20.2.2 $T \underset{k}{<} T'$ if

- $T \subseteq T'$ (This inclusion is among *multisets* of packets.)
- $\exists$ packet p s.t. $\text{mult}(p, T) < \text{mult}(p, T') \leq k)$ ($\text{mult}(p, T)$ denotes the multiplicity of p within the multiset T).

Lemma 20.3 *If α is valid, and T is a multiset of packets in transit after α has taken place, then either*

1. *$\exists$ k-extension $\alpha\beta$ such that the multiset of packets received by A^r in β is a submultiset of T, or*
2. *$\exists$ a valid execution $\alpha' = \alpha\beta$ such that*

   2.1. all packets received in β are sent in β,

   2.2. and $\exists$ a new multiset T' of packets in transit after α' such that $T \underset{k}{<} T'$.

Assume that this lemma is true. We then show that there is some valid α_∞, and a multiset T_∞ of messages in transit after α_∞ such that case (a) holds. As we already argued, the existence of such α_∞ and T_∞ leads to the desired contradiction.

For this we define two sequences α_i and T_i with $\alpha_0 = \alpha$ and $T_0 = T$. If condition a) does not hold for α and T (i.e.,for $i = 0$) we are in the situation of case (b). We then set $\alpha_1 = \alpha'$ and $T_1 = T'$. Generally, assuming that case (a) does not hold for α_i and T_i, we are then in case (b) and derive a valid extension α_{i+1} of α_i and a multiset T_{i+1} of packets in transit after α_{i+1} ($T_i \underset{k}{<} T_{i+1}$). But, by definition of the $\underset{k}{<}$ relation, the sequence $T_0 \underset{k}{<} T_1 \underset{k}{<} \ldots \underset{k}{<} T_i \underset{k}{<} \ldots$ can only have at most $k|P|$ terms. Its last term is the T_∞ we are looking after. ($|P|$ is the number of different possible packets, which is finite as we assumed that the packet alphabet and the size of the headers are bounded.)

Proof: (of Lemma 20.3)

Pick any m, and get a k-extension $\alpha\beta$ for m, by the k boundedness condition. If the multiset of packets received by A_r is included in T we are in case (a). Otherwise there is some packet p for which the multiplicity of *rec-pkt*(p) actions in β is bigger then mult(p, T). We then set $T' = T \cup \{p\}$. On the other hand, as the extension is k-bounded, the number of of these *rec-pkt*(p) is at most k so that mult$(p, T') \leq k$.

We now want to get a *valid* extension α' of α leaving the messages from T' in transit.

We know that there is a *send-pkt*(p) action in β (since all packets received in β were sent in β). Consider then a prefix $\alpha\gamma$ of $\alpha\beta$ ending with this *send-pkt*(P). (See Figure 20.12.)

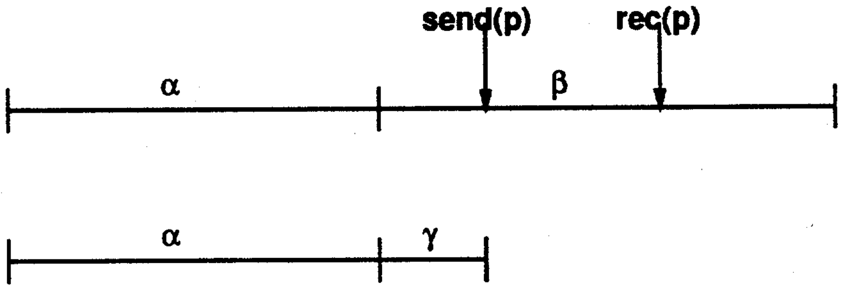


Figure 20.12: An extension of α with a $send - pkt(p)$ action

After $\alpha\gamma$ all messages from T' are still in transit. We want to extend $\alpha\gamma$ into a valid extension without delivering any packet from T'. To get validity we need to deliver m which is the only non delivered message without delivering a message from T'. We can achieve this because of the property of *fair* executions: there is a fair execution that does not deliver T' that eventually delivers m. ■

If the restriction on the number of packets needed to deliver a message is removed, it becomes possible to deliver messages reliably. The transmitter simply sends many copies of the packet, with the total number of packets sent increasing each time.

However, this greatly increases the time complexity of the algorithm.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 21: November 29

Lecturer: Nancy A. Lynch *Scribe: Andrew Chou*

21.1 Synchronous Algorithms

In this lecture, we will start studying synchronous algorithms. Synchronous algorithms are timing dependent: processes are assumed to take steps concurrently or in a round-robin fashion and communicate through message passing. The execution can be broken into a series of rounds, where in each round:

1. All processes send messages to all other processes. (Default messages can be sent if no message is required.)
2. All processes receive messages from all other processes.
3. Each process does internal computations and state changes.

I/O automata no longer form accurate models of synchronous systems unless we restrict the order of external steps in I/O automata, or we view the entire system as one large I/O automata with the round system imposed.

We will consider the two following problems:

1. Leader election,
2. Distributed consensus.

In this lecture, we will look at the distributed consensus problem.

21.2 Distributed consensus

We will begin by showing some impossibility results.

21.2.1 Gray's Two Generals Problem

The first problem we will look at is Gray's two generals problem. Two generals are planning an attack against a common objective from different sides. In order for the attack to be successful, both generals must attack simultaneously. If a general attacks alone, his army will be destroyed. Messages are sent via messengers, who may be killed en route.

Consider the following model: Let p_1 and p_2 be two processors that must agree on single bit. Processes cannot fail, but an unlimited number of messages can be lost. Consensus must be reached in a fixed number of rounds, r. (see Figure 21.1.)

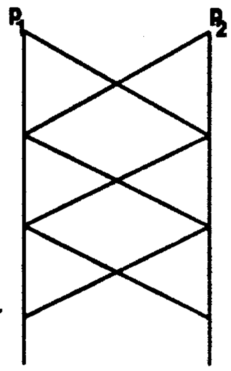


Figure 21.1: Pattern of message exchanges between p_1 and p_2

We want the following two conditions to hold:

Agreement: Both decide on the same value.

Validity: If both start with 0, then they both decide 0. If both start with 1 *and* all messages are delivered, then they both decide 1.

Claim 21.1 *No synchronous algorithm can solve Gray's two generals problem.*

Proof: The proof is by contradiction, using techniques similar to other impossibility proofs. Let α_1 be an execution where both p_1 and p_2 start at 1, and all messages are delivered. Without loss of generality, we can assume that the r rounds are completed, and consequently that $2r$ messages are sent: even if agreement is reached before, we can always pad the communication with extra messages. By validity, both processes must decided 1. Let α_2 be the same as α_1, except that the last message from p_1 to p_2 was not delivered. α_1 and α_2 looks the same to p_1, so p_1 decides 1. By agreement, p_2 must also decide 1. Now, let α_3 be α_2 with the last message from p_2 to p_1 removed. Using, the same argument as above, we see that p_1 and p_2 still decide 1. Now continue to remove messages, so that if n is even, α_n is the same as α_{n-1} with the last message from p_1 to p_2 removed, and if n is odd α_n is the same as α_{n-1} with the last message from p_2 to p_1 removed. For each α_i, both processes still decide 1.

We eventually arrive at the execution, α_{2r+1}. Here, both processes start at 1, no messages are delivered, and both decide 1. Now, let p_1 start at 0. To p_2, this looks the same as if p_1 started at 1. So, p_2 will decide 1. By agreement, p_1 decides 1. But, as no messages are delivered, p_1 cannot distinguish the situation with the situation where both p_1 and p_2 start at 0. In this case, they both decide 0, by the validity condition. We get a contradiction. ∎

21.2.2 Byzantine Agreement

Throughout this section, we will let n be the number of processes, f an upper bound on the number of faulty processes, and r the number of rounds in the algorithm that we consider.

In the Byzantine agreement problem that we now consider, the messages are passed in a reliable way, but the processes can be faulty: the adversary is stronger than in the General's problem.

We consider the following situation:

- All processes are connected.
- Messages are reliably delivered in each round.
- All processes start with initial values.
- Consensus among non-faulty processes must be reached by the end of the rth round, and the two following properties are satisfied:

  Agreement All non-faulty processes decide the same value.

  Validity If all non-faulty processes start with the same value v, they must decide v by the end of the protocol.

With no faulty processes, this problem is trivial. However, we want to be able to tolerate Byzantine faults. A faulty process is allowed to send any messages (including different messages to different processes). With an unbounded number of faults, it is impossible to say anything. Intuitively, there should be an upper bound (in terms of n) on the size of f for which Byzantine agreement is achievable.

We begin by giving a heuristic showing that, for $n = 3$, the potential presence of a single faulty process precludes achievability of Byzantine agreement. (i.e. for $n = 3$ and $f = 1$, Byzantine agreement is not achievable.) Consider the case of three processes: A, B, and C. Because of the 1-resilience of the system, if B and C hold an initial value 0 and are honest, they will decide 0 whatever A sends them: they can, by default, consider that A is faulty.

In the same way if two (honest) processes hold 1 as their initial value, they must decide 1 at the end of the protocol. Consider now the situation where A and B are honest and hold initial values 1 and 0 (respectively). Assume that C is faulty and behaves to A exactly as if it were holding 1 and behaves to B exactly as if it was holding 0. At this point, A will of course detect that B or C is faulty, but will by *unable* to conclude. (Recall that by assumption, at most one process is faulty.) But this means that A *cannot* decide: in effect, it can decide 1 only if concludes that B is faulty, and it can decide 0 only if it concludes that C is faulty.

This result will be established more formally in the next lecture.

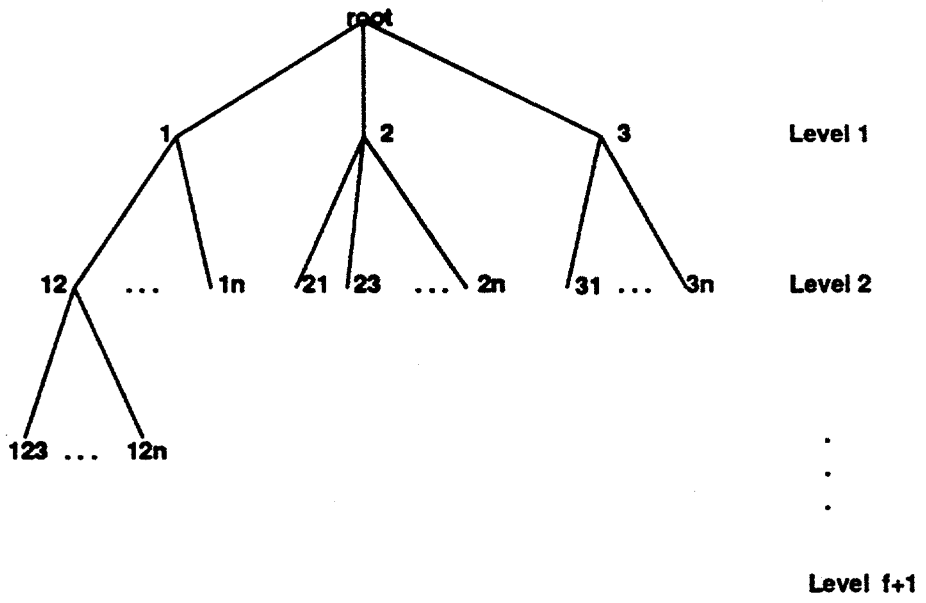


Figure 21.2: Tree representing the paths of messages between processes

21.2.3 Algorithm for Byzantine Agreement

We now present the algorithm [LaPeSh80] by Lamport, Pease, and Shortak (in short LPS's algorithm) for Byzantine agreement. It satisfies the following correctness statement:

Theorem 21.2 *If $n \geq 3f+1$ LPS's algorithm solves Byzantine agreement in $f+1$ rounds.*

To describe and analyze the algorithm, we first construct the tree of Figure 21.2.

It has a $f+2$ levels of (the root is at level 0). A node at level k, $k = 1, 2, 3, \ldots f+1$, is labeled $i_1 i_2 i_3 \ldots i_k$, so that its father is labeled $i_1 i_2 i_3 \ldots i_{k-1}$, its grandfather is labeled $i_1 i_2 i_3 \ldots i_{k-2}$, and so on.

We will extend our notation to $i_0 = root$, which corresponds to $k = 0$. Any node $i_1 \ldots i_k$ at level k ($k = 0, 1, 2, \ldots, f$) has degree $n-k$. Its children are labeled $i_1 i_2 \ldots i_k j$ for $j \in \{1, 2, 3, \ldots, n\} - \{i_1, i_2, i_3, \ldots, i_k\}$.

We are now ready to begin describing the algorithm. This algorithm works for $n \geq 3f-1$ and requires $r = f+1$ rounds. The protocol consists of two phases: a *down*-phase and an *up*-phase. We begin describing the *down*-phase.

The *down*-phase is divided into r rounds.

- In the first round, each process i sends to all processes j (including itself) the value b_i that it holds along with its name: $\forall$ processes i and j, i sends to j the message $(b_i; (i))$.
- In the subsequent rounds k ($k = 2, \ldots, r$), each process i relays to all processes j all the "well-formed" messages it has received in round $k-1$. Before doing so, it affixes

its id to the second field of the message. Hence, this second field holds the path $i_1 \dots i_k$ that the path the message went through up to round k.

By "well-formed" we mean two things:

- In round k process i relays only messages that are of the form $(b; (i_1 \dots i_{k-1}))$, where b is a value and $i_1 \dots i_{k-1}$ is a sequence of distinct process id's.
- Process i relays only messages that it has never seen: formally i relays only messages such that $i \neq i_1, i \neq i_2, \dots, i \neq i_{k-1}$.

The preceding can be encoded as: In round k: $\forall$ process i, $\forall$ message m received by i in round $k-1$,

if m is of the form $(b; (i_1 \dots i_{k-1}))$ with $i \neq i_1, \dots, i_{k-1}$,
then $\forall j$ such that $j \neq i_1, \dots, i_{k-1}, i$; process i transmits $(b; (i_1 \dots i_{k-1}, i))$ to j.
else: do nothing.

The tree constructed in Figure 21.2 is exactly the tree of the paths of the messages sent during the protocol. For instance, a path ijk says that k transmitted (to you) what j transmitted (to k), which is what i transmitted (to j). The "transmitted" in the previous sentence includes the possibility of a faulty message if the process transmitting it is itself faulty.

For each message m, the protocol ensures that its second field contains the path p it used so far: $m = (b; p)$. In the following, for any path $p = (i_1, \dots, i_k)$ we let $|p|$ denote the length k of p. We also say that p ends in i_k.

Note that each path of length (at most) $f + 1$ is represented exactly once in the tree. Now we can think that *each* process i is provided with its *own* copy of the tree. For a path p of the tree, we define $value_i(p)$ to be the value that i receives along the path p. This means that in round k, i receives all the messages $(value_i(p); p)$, where $|p| = k$ and p is well formed. (Note that we do not preclude the fact that $value_i(p)$ could be empty: this could happen if the path p was containing a faulty process halting its transmission.) Hence, even though two processes i and j hold identical copies of the tree, for any path p, $value_i(p)$ and $value_j(p)$ need not be equal. Nevertheless, the following obvious fact holds:

Lemma 21.3 *If i, j, and k are all non-faulty, then $value_i(p) = value_j(p)$ for every path ending with k.*

Proof: k is a honest guy and sends the same thing to everyone! ■

We now describe the *up*-phase. In the up-phase, *no* communication takes place among different processes; instead, each (non-faulty) process performs local computations based on its tree and the value it obtained along all its paths. Even though there is no communication, we will still describe for convenience the up-phase in rounds. The aim of the up-phase is to compute inductively (in a backwards way) $newvalue_i(p)$. For all paths p, the decision value of i will be $newvalue_i(\text{root})$.

Every $value_i(p)$ missing from the down-phase (due to faulty processes) is set to any arbitrary value.

- Base Case: $\forall p$ such that $|p| = f+1, newvalue_i(p) = value_i(p)$
- Inductive step: For $k = 2, \ldots f+1$. Assume round $k-1$ is over. $\forall$ path p such that $|p| = f+2-k$, compute:

  $newvalue_i(p) :=$ majority value of $\{newvalue_i(p.j);\ (p.j)$ is a child of p in the tree$\}$

  If no majority exists, $newvalue_i(p)$ is set by default to 0.
- Decision: i decides $newvalue_i(\text{root})$.

We now establish the correctness of this protocol. *Validity* is easy to prove: Suppose that all processes start with the same value v. Then, all non-faulty processes send v at the first step, so that $value_i(j) = v$. Then, $newvalue_i(j) = v$. So, by majority rule, $newvalue_i(\text{root}) = v$. We now turn to the proof of agreement.

Lemma 21.4 *Suppose that $p = (j_1, \ldots, j_{f+2-k})$ is a path ending with the label j_{f+2-k} of a non-faulty process. Then there is a value v such that $value_i(p) = newvalue_i(p) = v$ for all non-faulty process i.*

Proof: We apply backwards induction on the length $|p|$ of the path p.

Base case, $k = 1$: p is a leaf of the tree (i.e. $|p| = f+1$). Since j_{f+1} is non-faulty, it sends in the last round of the down-phase the same value $value_i(p)$ to all process i. Hence, in round 1 of the up-phase, any non-faulty process i uses this value (by Lemma 21.3) to set $newvalue_i(p)$.

Induction step, for $k = 2, \ldots, f+1$: Assume round $k-1$ is over. Consider a path p such that $|p| = f+2-k$ and p ends with the label of a non-faulty process j_{f+2-k}. Since j_{f+2-k} is non-faulty in round $f+3-k$ of the down-phase, it sent the *same* value $value_i(p)$ to all process i. Let v denote this common value: $v \overset{\text{def}}{=} value_i(p)$.

As a consequence, all the non-faulty processes, l, sent the same $value_i(pl) = v$ to all processes i. As noted previously, at label $f+2-k$ the tree has degree $n-(f+2-k)$. (This degree represents the number of processes that process j_{f+2-k} communicates with in round $f+3-k$ of the down-phase.) But,

$$\begin{aligned} n-(f+2-k) &= n+k-2-f \\ &\geq n-f \text{ for } k \geq 2, \\ &\geq 2f+1 \text{ since } n \geq 3f+1 \text{ by assumption.} \end{aligned}$$

Hence, a *majority* of these processes l are non-faulty. By induction hypothesis, we then have that for all non-faulty process i, $newvalue_i(pl) = value_i(pl) = v$. Thus,

$$\begin{aligned} newvalue_i(p) &\stackrel{\text{def}}{=} \text{majority value of } \{newvalue_i(pl) \mid pl \text{ is a child of } p \text{ in the tree}\} \\ &= v \\ &\stackrel{\text{def}}{=} value_i(p). \end{aligned}$$

■

Definition A node, p, is *common* if all non-faulty processes have the same $newvalue(p)$ value.

Definition A *frontier* of a tree is a set of nodes containing at least one node on every path in the tree.

Definition A *common* frontier is a frontier consisting of common nodes.

Lemma 21.5 *The tree contains a common frontier.*

Proof: Consider the $f+1$ non-root nodes on any path. We can write them i_1, $i_1 i_2$, $i_1 i_2 i_3$, $\ldots$, $i_1 i_2 \ldots i_{f+1}$ where $i_i \neq i_j$ for $i \neq j$. One of these $f+1$ i_j is not faulty. Call it i_{j_0}. Then node $i_1 i_2 \ldots i_{j_0}$ is common. ■

Lemma 21.6 *Let p be a path (with our conventions a path corresponds to a node). If there is a common frontier in the subtree rooted at p, then p is common.*

Proof: Apply reverse induction again.

Base case: p is a leaf - trivial.

Inductive step: Assume this property shown for the nodes p of length $|p| = k+1$ and consider a node p of length $p = k$. By hypothesis, p has a common frontier that we denote $\mathcal{F}$. Consider any child pl of p. If the frontier $\mathcal{F}$ goes through pl, then, of course, pl is common. Otherwise, $\mathcal{F}$ induces a frontier on the tree rooted at pl. Then, by the induction hypothesis, we deduce that pl is common. Thus, all children of pl of p are common. ■

Corollary 21.5 and 21.6 immediately imply that:

Corollary 21.7 *Root is common.*

This finishes the proof that the algorithm satisfies *agreement.*

21.2.4 Non-Byzantine failures

A much simpler tree-based algorithm exists for $n \geq f+1$ with stopping faults. (We allow a process to stop in the middle of its broadcast within a round.) It is based on the consideration of the same tree and on the sending of the same messages $value_i(p)$ as in the previous algorithm. Let $V_i = \{v | v = value_i(p) \text{ for some } p\}$. The difference with the previous algorithm is that the result is determined to be any standard member of that set (e.g., the smallest number).

Claim 21.8 *Agreement holds, that is $V_i = V_j$ for all non-faulty i,j.*

Proof: Suppose $v \in V_i$ so that $v = value_i(p)$ for some p. If $|p| \leq f$, then $|pi| \leq f+1$, and $value_j(pi) = v$ (i.e., i relays the value v). If $|p| = f+1$, then there is *some* non-faulty process on p so that $p = qlr$, where q,r are paths and l is a non-faulty process. Then, $value_j(ql) = v$. ■

Validity holds, since if all processes started at v, this is the only value that gets anywhere.

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 22: December 4

Lecturer: Nancy A. Lynch *Scribe: Robert Blumofe*[24]

22.1 Lower Bounds for Agreement

A process with Byzantine faults is like a worst-case adversary. We have seen that agreement, even in the case of Byzantine faults, can be solved by a deterministic algorithm. The algorithm we saw used $n \geq 3f + 1$ processes and $f + 1$ synchronous rounds of communication where f is the maximum number of faults. Later we will see if this problem can be solved with fewer than $f+1$ rounds of communication. But first we ask, can we solve this problem with fewer than $3f + 1$ processes?

22.1.1 Number of Processes for Byzantine Agreement

No. 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 of last lecture. A formal proof of this is given in [PeaseSL80]; this proof follows that in [FischerLM86].

Lemma 22.1 *Three processes cannot solve Byzantine agreement in the presence of one fault.*

Proof: Working by contradiction we assume they can, using a protocol $\mathcal{P}$. Then there exist three processes, A, B, and C which, when arranged in a system and given arbitrary inputs will satisfy the Byzantine agreement conditions even if one process malfunctions.

We could take two copies of each process, and arrange them into a 6-process system S as shown in Figure 22.1.

When configured in this way, the system appears to every process as if it is configured in the original three-process system. We then define a protocol $\mathcal{P}_S$ on S: $\mathcal{P}_S$ is characterized by the fact that each process is running $\mathcal{P}$ as if the universe consisted only on itself and its two neighbors. We will show that $\mathcal{P}_S$ exhibits a contradictory behavior.

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

[24]Based on lecture notes from 1988 scribed by Jeff Fried, Jeff Palmucci, and George Varghese.

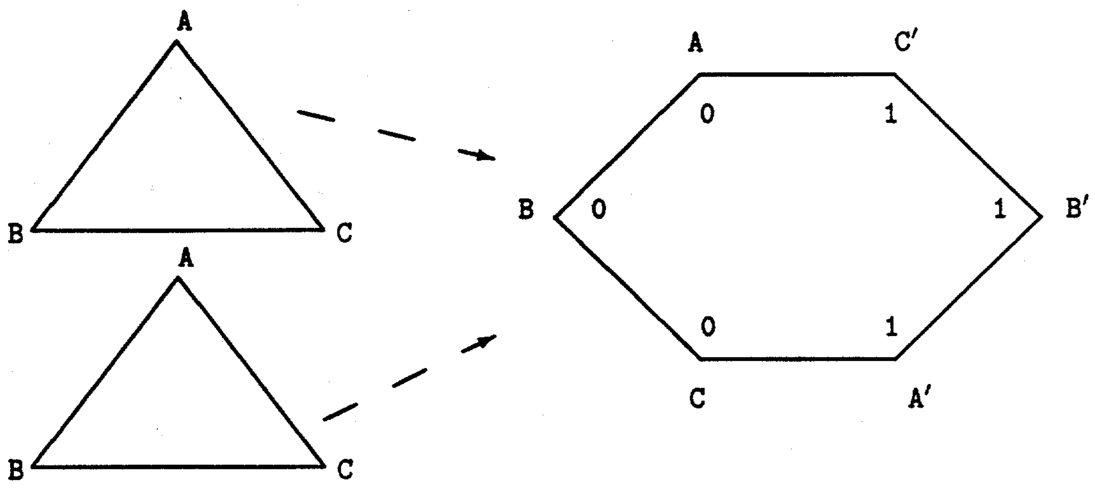


Figure 22.1: Impossibility result for Byzantine agreement on three processes with one fault.

Agreement on three processes, so B and C must eventually agree on 0 in the three-process system. Since the six-process system S appears identical to B and C, they will eventually agree on 0 in S as well.

Next consider the processes C-A'-B'-C'. By similar reasoning, A' and B' 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). However this is impossible since we just saw that C must decide 0 and A' must decide 1. 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 now use this result to show that Byzantine agreement requires $n \geq 3f + 1$ processes to tolerate f Byzantine faults [LamportPS82]. We will do this by showing how an $n \leq 3f$ process solution which can tolerate f Byzantine failures can be used to construct a 3 process solution which can tolerate a single Byzantine failure; this of course contradicts the above lemma.

Theorem 22.2 *There is no solution to the Byzantine agreement problem on n processes in the presence of f Byzantine failures, when* $1 < n \leq 3f$.

Proof: Assume there is a solution for Byzantine agreement with $3 < n \leq 3f$. (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 $1 \leq s \leq f$. Let the three new processes be p_1, p_2, and p_3, and let each p_i simulate the original processes in P_i. Each process p_i keeps track of the states of all the original processes in P_i, assigns its own initial value to every member of P_i, and simulates the steps of all the processes in P_i as well as the messages between the processes in P_i. Messages from processes in P_i to processes in another subset are sent from p_i to the process simulating that subset. When a simulated process in P_i decides on a value v then p_i can decide on the value v.

To see that this is a correct 3-process solution we reason as follows. Only one of p_1, p_2, p_3 is allowed to be faulty, and each simulates between 1 and f original processes, so the simulation contains no more than f simulated faults. The n-process simulated solution then guarantees that the simulated processes satisfy agreement, validity, and termination. Any process p_i which is non-faulty simulates only non-faulty processes. Hence if a process in P_i decides on a value v, all other process in P_i must decide v, so that p_i can safely use v as its decision. Thus the validity, agreement, and termination of the n-process simulation carries over to the 3-process system. This is a contradiction. ∎

22.1.2 Byzantine Agreement in General Graphs

We have shown that Byzantine agreement can be solved with n processes and f faults, where $n \geq 3f + 1$. 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 reach agreement. Similarly, if removing f nodes can disconnect the graph, it should also be impossible to reach agreement with f 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 22.2 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

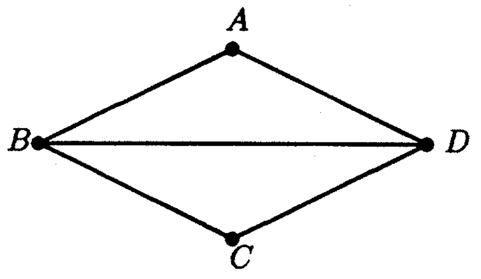


Figure 22.2: A graph G with conn$(G) = 2$.

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 22.3 *It is possible to solve Byzantine agreement on a graph, G, with n nodes and f faults if and only if*

1. $n \geq 3f + 1$, *and*

2. *conn*$(G) \geq 2f + 1$.

Proof: We already know that $n \geq 3f + 1$ 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$(G) \geq 2f + 1$. (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 $2f+1$-connected, there are at least $2f+1$ node disjoint paths between any two nodes. We can simulate a direct connection between these nodes by sending the value along each of the $2f+1$ paths. Since only f 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 saw in Lecture 21 an algorithm of Lamport, Pease and Shortak that solves byzantine agreement in this situation.

We now prove the *only if* direction of the connectivity argument. The argument that Byzantine agreement is not possible if $\text{conn}(G) \leq 2f$ is a bit more intricate. We will first take $f = 1$, for simplicity.

Assume there exists a graph, G, with $\text{conn}(G) \leq 2$ which can solve Byzantine agreement with one fault. Two points in G can disconnect the graph. The graph in Figure 22.2 can be generalized to any graph with a connectivity of 2 by replacing A and C with arbitrary graphs. To keep our argument simple, however, we will consider A and C to be single nodes. We can construct a graph $\mathcal{C}$ by "rewiring" two copies of graph G, as shown in Figure 22.3. Each process in $\mathcal{C}$ behaves as if it was the same-named process in Figure 22.2 with the input denoted by the subscript.

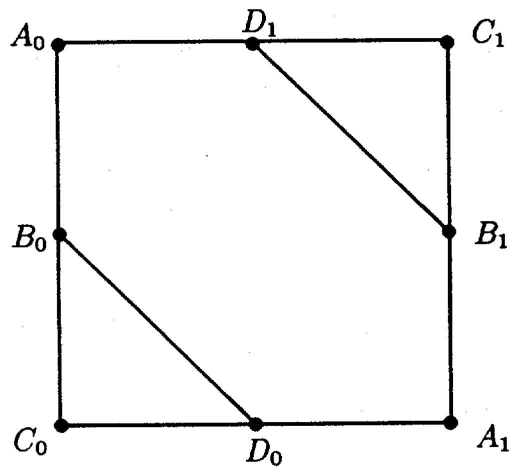


Figure 22.3: Graph $\mathcal{C}$, made by "rewiring" two copies of G.

Consider the behavior of the processes outlined in Figure 22.4, and the corresponding behavior in 22.5, where F is a faulty process. By the fault-axiom, the outlined processes cannot tell the difference between Figure 22.4 and Figure 22.5. Therefore, by the validity property, these processes must all decide 0.

Now consider Figure 22.6 and the corresponding 4-nodes situation of Figure 22.7. By the same argument, all the outlined processes are required to decide 1.

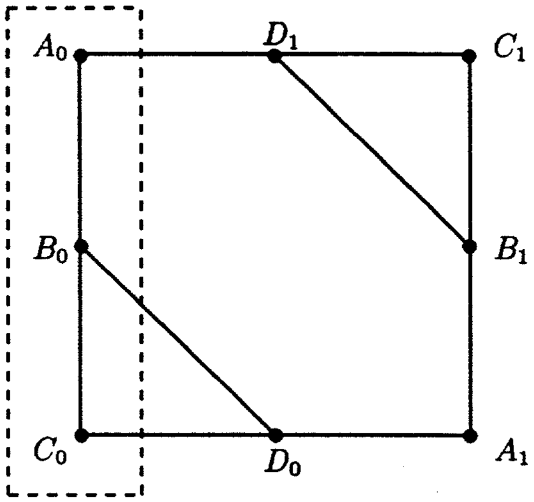


Figure 22.4: A set of processes in $\mathcal{C}$.

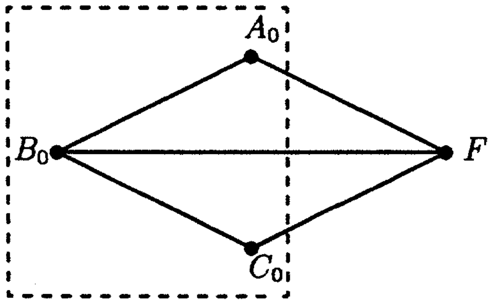


Figure 22.5: A configuration (with F faulty) that the outlined processes cannot distinguish from Figure 22.4.

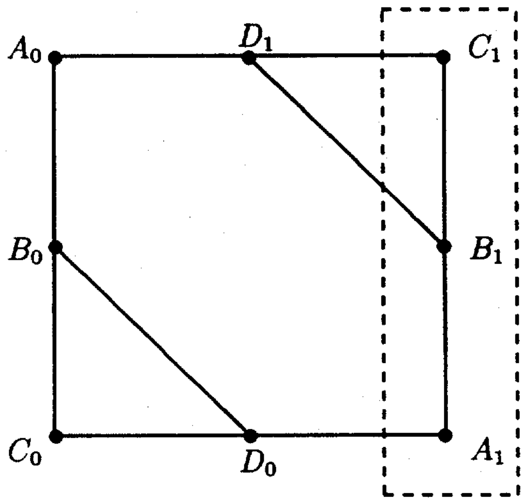


Figure 22.6: A set of processes in $\mathcal{C}$.

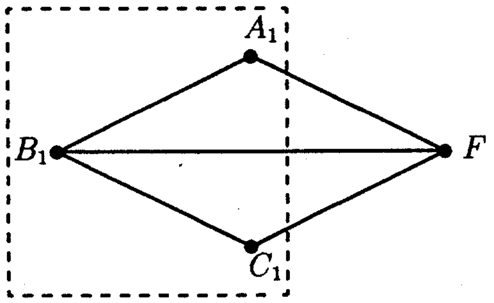


Figure 22.7: A configuration (with F faulty) that the outlined processes cannot distinguish from Figure 22.6.

Finally, consider Figure 22.8, which is corresponding 4-nodes situation of Figure 22.9. Since only F is faulty, the agreement condition requires that the outlined processes decide on the same value.

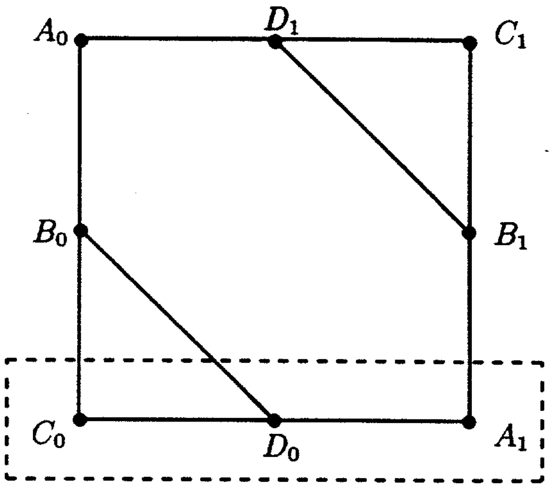


Figure 22.8: A set of processes in $\mathcal{C}$.

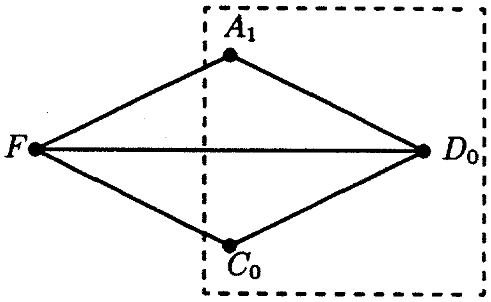


Figure 22.9: The non-faulty processes must agree, giving us a contradiction.

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 $f = 1$.

To generalize the result to $f > 1$, we use the same diagrams, with B and D replaced by graphs of at most f nodes each and A and C by arbitrary graphs. Again, removing B and D disconnects A and C. The edges of Figure 22.2 now represent all possible edges between A, B, C, and D. ∎

22.1.3 Weak Byzantine Agreement

Lamport considered weakening the requirements for Byzantine agreement (still considering Byzantine faults) by changing the validity requirement.

- **Validity:** If all processes start with value v and no faults occur, then v is the only allowable decision value.

Previously we required that even if there were faults, if all processes started with v then all non-faulty processes must decide v. Now they are only required to decide v in the case of no failures. This weakened restriction corresponds to the requirements of the database commit problem since we only require commitment in the case of no faults.

Lamport tried to get better algorithms for weak Byzantine agreement than for Byzantine agreement but failed. Instead he got the impossibility result:

Theorem 22.4 *$n \geq 3f + 1$ processes and conn$(G) \geq 2f + 1$ are needed even for weak Byzantine agreement.*

We will just show that three processes cannot solve weak Byzantine agreement with one fault; the test of the proof follows as before.

Suppose there exists three processes A, B, and C which can solve weak Byzantine agreement with one fault. Let G_0 be the execution where all three processes start with zero, no failures occur, and therefore all three processes decide zero. Let G_j, $1 \leq 3$, be the same execution except that j of the three processes start with a one. Let k be strictly larger than the number of rounds in executions $G_0, \ldots, G_3$.

We now create a new system S with at least $4k$ nodes by pasting enough copies of the sequence A-B-C into a ring. Let half the ring start with value zero and the the other half start with one. Now consider the resulting execution. By arguing as we did before, any two consecutive processes must agree; therefore, all processes in S must agree. Assume (without loss of generality) that they choose one. Each process will reach this decision in fewer than k rounds. But there is some process in the middle of the half of S which started with zeros and is more than k steps away from any processes which started with one. This processes will behave exactly as it did in G_0 and therefore must decide zero. This is a contradiction.

22.1.4 Number of Rounds with Stopping Faults

We now turn to the question, can Byzantine agreement be solved in fewer than $f+1$ rounds. Once again the answer is no. As if that's not bad enough, we will see that at least $f + 1$ rounds are required even to simply tolerate f stopping faults. This is true regardless of how big n is (though we will assume $n \geq f + 2$).

The Model

Each non-faulty process is modeled by a deterministic 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 all other processes. Each process then computes its new state based on all received messages before the next round starts.

Each process starts with an initial value. Eventually a process may write "decide V" in its state, but for simplicity, we assume a process continues executing forever even after it decides.

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 22.10.

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. Let ρ be a run, and denote by exec(ρ) 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 f faulty processes in a run.

The Case of One Failure

We will now show that two rounds are required to handle a single stopping fault.

We do this by contradiction, so assume a 1-round algorithm exists. We will construct a chain of executions such that the first execution must lead to a 0 decision, the last execution must lead to a 1 decision, and for any two consecutive executions in the chain there will be some non-faulty process to which both executions look the same. This of course is an impossible situation since if two executions look the same to a non-faulty process, that process must make the same decision in both executions, and therefore by the agreement

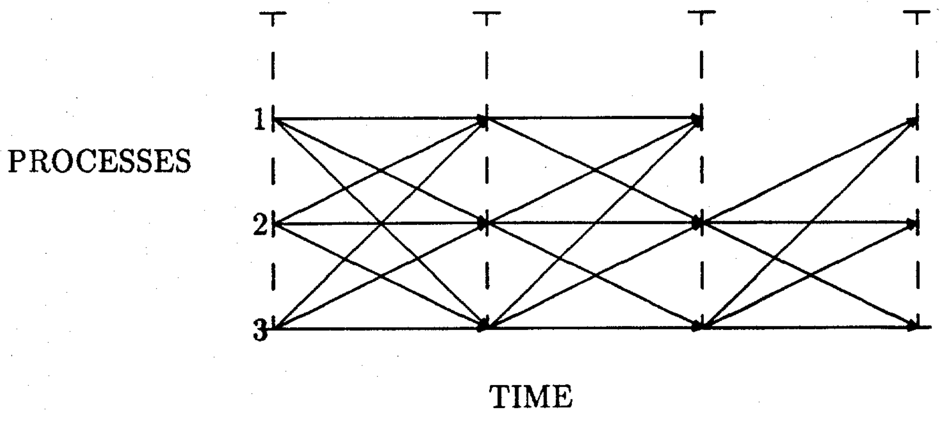


Figure 22.10: Example of a communication pattern.

property both executions must have the same decision value. Thus every execution in the chain must have the same decision value which is impossible since the first execution must decide 0 and the last must decide 1.

We start the chain with the execution determined from all zeros as input and the complete communication pattern as in Figure 22.11. This execution must decide 0.

Starting from this execution, form the next execution by letting process p_1 fail at the beginning and miss sending a message to itself. These first two executions look the same to all other processes, and at least one of them must be non-faulty ($n \geq f + 2$).

Form the next execution by removing the message from p_1 to p_2. These two executions look the same to all processes except p_1 and p_2, and again at least one of these $n-2$ processes must be non-faulty. Next we remove the message from p_1 to p_3, and these two look the same to all processes except p_1 and p_3. We continue in this manner removing one message at a time so for every consecutive pair of executions, they look the same to some non-faulty process.

Once we have removed all the messages from p_1, we form the next execution by changing p_1's input value from 0 to 1. Of course these two executions will look the same to every process except p_1 since p_1 sends no messages. Now we can add the messages back in one by one, and again for every consecutive pair of executions, they will look the same to some non-faulty process.

We can repeat this construction for p_2: removing p_2's messages one-by-one, changing p_2's input value from 0 to 1, and then adding p_2's messages back in. Repeating this construction

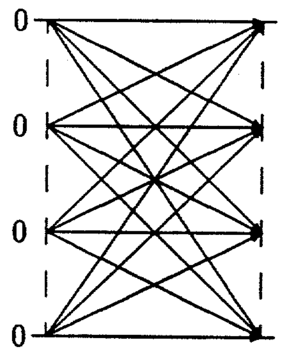


Figure 22.11: A run which must decide zero.

for $p_3, \ldots, p_n$ we end the chain with the same execution we started with except with all ones as input. This execution must decide 1. Thus we have produced a chain as claimed, and this is a contradiction.

The Case of Two Failures

We will now show that two rounds are not sufficient to handle two stopping failures. This is done by forming a chain as we did for the case of one fault and one round. We start with the execution determined by all zeros as input and the complete communication pattern; this execution must decide 0.

To form the chain we want to work toward killing p_1 at the beginning. When we were only dealing with one round we could kill messages from p_1 one-by-one. Now, if we delete a first-round message from p_1 to q in one step of the chain, then it is no longer the case that the two executions must look the same to some non-faulty process. This is because in the second round q could inform all other processes as to whether or not it received a message from p_1 in the first round, so that at the end of the second round the two executions can differ for all non-faulty processes.

We solve this problem by using several steps to delete the first-round message from p_1 to q, and by letting q be faulty too (we are allowed two faults). We start with an execution in which p_1 sends to q in the first round and q sends to every process in the second round. Now we let q be faulty and remove second-round messages from q until we have an execution in which p_1 sends to q in the first round and q sends no messages in the second round. Next we can remove the first-round message from p_1 to q, and clearly these two executions will only look different to p_1 and q. Now we replace second-round messages from q one-by-one until we have an execution in which p_1 does not send to q in the first round and q sends to all in the second round. This achieves our goal of removing a first-round message from p_1

while still maintaining that for every consecutive pair of executions, they look the same to some non-faulty process.

Now we can remove first-round messages from p_1 one-by-one until p_1 sends no messages, change p_1's input from 0 to 1, and replace p_1's messages one-by-one. Repeating this for $p_2,\ldots,p_n$ as before gives the desired chain.

The General Case

We now consider the general case so we can have up to f faulty processes in a run. Assume $f \geq 1$ and suppose we have an f round protocol.

If ρ and ρ' are runs with p non-faulty in both then we write $\rho \overset{p}{\sim} \rho'$ to mean that exec(ρ) and exec(ρ') look the same to p through time f (same state sequence and same messages received). We write $\rho \sim \rho'$ if there exists a process p which is non-faulty in both ρ and ρ' such that $\rho \overset{p}{\sim} \rho'$. We write $\rho \approx \rho'$ for the transitive closure of the $\sim$ relation.

Every run ρ has a decision value denoted by dec(ρ). If $\rho \sim \rho'$ then dec(ρ) = dec(ρ') since ρ and ρ' look the same to some non-faulty process. Thus if $\rho \approx \rho'$ then dec(ρ) = dec(ρ').

Now let F_ρ be the set of failures in run ρ, and consider the following lemma.

Lemma 22.5 *Let ρ and ρ' be runs. Let p be a process. Let k be such that $0 \leq k \leq f-1$. If $|F_\rho \cup F_{\rho'}| \leq k+1$ and ρ and ρ' only differ in p's failure behavior after time k then $\rho \approx \rho'$.*

Proof: We prove this lemma by reverse induction on k as k goes from $f-1$ to 0.

As the basis we have $k = f-1$. In this case ρ and ρ' agree up to round $f-1$ so consider round f. If ρ and ρ' don't differ at all then we're done, so suppose p is faulty in at least one of ρ or ρ' and the total number of other processes that fail in either ρ or ρ' is no more than $f-1$. Consider two processes, q and r, which are non-faulty in both ρ and ρ'. (They exist since $n \geq f+2$.) Let ρ_1 be the same as ρ except that p sends to q in round f of ρ_1 exactly if it does in ρ'. As an example see Figure 22.12. We have $\rho \overset{r}{\sim} \rho_1$ and $\rho_1 \overset{q}{\sim} \rho'$ so $\rho \approx \rho'$.

We now turn to the inductive step. We want to show that the lemma is true for k satisfying $0 \leq k < f-1$ assuming it is true for $k+1$. Executions ρ and ρ' agree up to round k. If they also agree up to round $k+1$ then we can apply the inductive hypothesis and we are done, so assume process p fails in round $k+1$, and assume (without loss of generality) that p fails in ρ.

Let ρ_i, for $1 \leq i \leq n$, be the same as ρ except that p sends to $p_1,\ldots,p_i$ at round $k+1$ of ρ_i exactly if it does in ρ'. Each ρ_i has no more than $k+1$ failures (since ρ has no more than $k+1$ failures). We claim that $\rho_{i-1} \approx \rho_i$ for $1 \leq i \leq n$ (defining $\rho_0 = \rho$), and we now proceed to verify this claim.

Executions ρ_{i-1} and ρ_i differ at most in what p sends to p_i at round $k+1$. Let ρ_i' be the same as ρ_{i-1} except that p_i sends no messages after round $k+1$ in ρ_i'. Similarly let ρ_i'' be the same as ρ_i except that p_i sends no messages after round $k+1$ in ρ_i''. This situation is

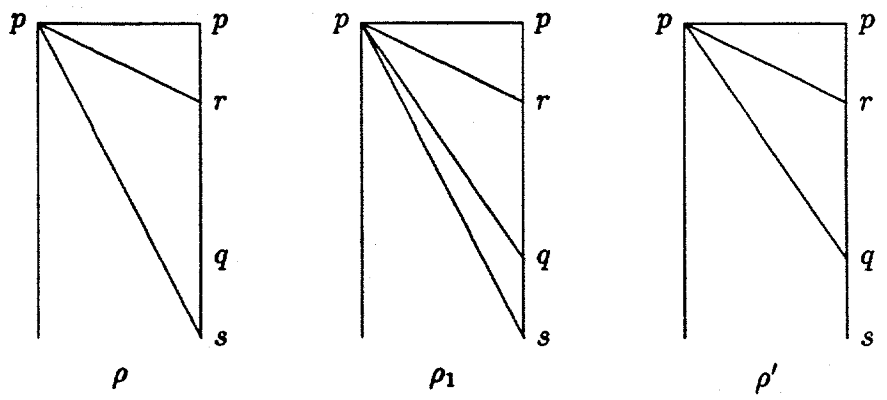


Figure 22.12: Example of construction used to prove base case of Lemma 22.5.

illustrated in Figure 22.13. By the inductive hypothesis we have $\rho_{i-1} \approx \rho'_i$ and $\rho_i \approx \rho''_i$. Also $\rho'_i \sim \rho''_i$ since both look the same to any non-faulty process (other than p_i). Thus $\rho_{i-1} \approx \rho_i$ as claimed.

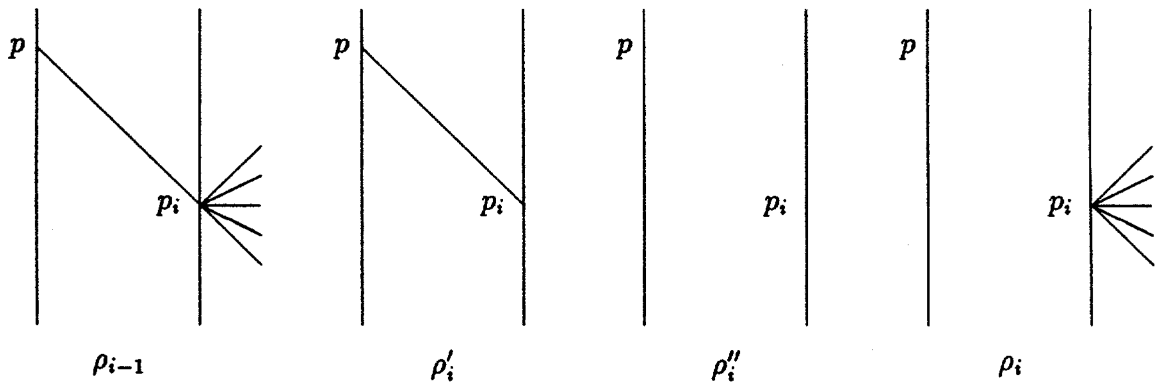


Figure 22.13: Example of construction used to prove inductive step of Lemma 22.5.

We now have $\rho = \rho_0 \approx \rho_n$. Also, ρ_n is the same as ρ' up through round $k+1$ so we can apply the inductive hypothesis to get $\rho_n \approx \rho'$. Thus we have $\rho \approx \rho'$ as needed. ∎

To get our result we want to consider the case $k = 0$ in Lemma 22.5. This says that if runs ρ and ρ' have only one failure between them and differ only in the failure behavior of that one process then $\rho \approx \rho'$. We use this as follows.

Let ρ_0 and ρ'_0 be two runs both starting will all zeros as input; in ρ_0 no process fails, and in ρ'_0 process p_1 fails at the beginning and sends no messages. There is only one failure between them and they only differ in p_1's failure behavior, so by the lemma, $\rho_0 \approx \rho'_0$.

Now let ρ''_0 be the same as ρ'_0 except p_1 has a 1 as input instead of 0. Since p_1 sends no messages in ρ'_0 this change cannot be seen by any process except p_1, so $\rho'_0 \sim \rho''_0$. Now let ρ_1 have the same input as ρ''_0 except there are no failures in ρ_1. Again the lemma says $\rho''_0 \approx \rho_1$. Therefore $\rho_0 \approx \rho_1$.

We continue this letting ρ_i be the execution with no failures and input defined by having the first i processes get 1 as input and the others getting 0. We have $\rho_{i-1} \approx \rho_i$, so $\rho_0 \approx \rho_n$, and therefore $\mathrm{dec}(\rho_0) = \mathrm{dec}(\rho_n)$. But this is a contradiction since ρ_0 has all zeros as input with no failures and hence must decide 0 whereas ρ_n has all ones as input with no failures and hence must decide 1.

Thus we have the following theorem.

Theorem 22.6 *Any agreement protocol requires $f+1$ rounds to handle f stopping failures.*

6.852 Distributed Algorithms **Fall Semester, 1990**

Lecture 23: December 11

Lecturer: Nancy A. Lynch *Scribe: Mary Ellen Zurko*[25]

23.1 Multivalued vs. Binary Byzantine Agreement

One way to agree on a single bit value from a set of values V is to treat this value as a bit string, and use a bit Byzantine Agreement algorithm A_{bit} to agree on the value bit by bit. But this is a very slow process.

23.1.1 Turpin & Coan's Algorithm

In their algorithm, Turpin & Coan use the fact that, in the problem of distributed byzantine agreement on a value v drawn from a large value set V, an economy in communication can be achieved by breaking the algorithm into two steps:

1. Solve BA on bits only,

2. Use the result as a subroutine to reach agreement on a value in V.

This breakdown indicates that the problem of getting a consensus is the most important part of the algorithm.

The algorithm, is similar to Ben-Or's algorithm: it just uses some few extra rounds to take into account the fact that the value to agree upon is not simply a bit. We assume that $n \geq 3f + 1$.(As usual n is the number of processes, and f is the number of faulty ones.). Every process has an initial value x.
Note that the default value might not be in the original value set.

23.1.2 Correctness

Termination

Termination in $f + 3$ rounds is obvious: the 2 rounds of the beginning of the previous algorithm and at most $f + 1$ rounds for the bit subroutine.

[25]Based on lecture notes from 1988 scribed by George Verghese

```
Round 1:
        Broadcast $(x)$
        if, in set of messages received, there are $\geq n - f$ for some $v$
                then $x \leftarrow v$
                else $x \leftarrow nil$
Round 2:
        Broadcast $(x)$
        Let $v =$ value with most occurrences among those received in Round 2
                (break ties arbitrarily)
        Define $m =$ number of occurrences of $v$
        then $x \leftarrow v$
        if $m \geq n - f$
                then $vote \leftarrow 1$
                else $vote \leftarrow 0$

Call the BA bit-subroutine with $vote$
If result = 1, choose $x$ (result saved after round 2)
If result = 0, choose some predetermined default
```

Figure 23.1: Turpin and Coan's Algorithm

Validity

If all processes start with a value w, then all nonfaulty processes get $\geq n - f$ w's in round 2, and they vote 1. The validity for the BA subroutine says they must agree on 1, and all nonfaulty processes will chose w.

A technicality: faulty processes could begin the BA with 0's, but it's just as if they started with 1's and lied about their initial value, so the nonfaulty processes are still required to decide 1.

Agreement

If the BA subroutine returns 0, agreement is clear (the default value is chosen).

In the case where the subroutine decides 1, we must argue that then every nonfaulty process chose the same x after the invocation of the bit protocol.

Lemma 23.1 *There is at most one value, x, that is ever sent in round 2 messages by nonfaulty processes.*

*Proof:*Let x be a value that is broadcasted in round 2 by some process p. This process must have received at least $n-f$ messages v in round 1, so that al other non faulty processes must have received $n-2f$ messages v and can have received only $2f$ messages w for any other w. But $2f < n-f$ so that a non faulty process will not broadcast w in round 2. ∎

If all processes agree on 1 via the subroutine, at least one nonfaulty process had to start with 1 as its input to the BA subroutine, by the validity of the subroutine. In fact, by a similar argument as the one of Lemma 23.1, some nonfaulty process must have started with 1. This process got $\geq n-f$ round 2 messages with value v (the v whose existence is ascertained in Lemma 23.1) so that all other processes received at least $\geq n-2f\,v$ messages in round 2. Since all the nonfaulty processes could only broadcast this value v in round 2, (by Lemma 23.1), all other value w received can only have been sent by some of the f faulty processes. But, as $f < n-2f$, this value w occur less often then v so that v is the value chosen at the end of the protocol by all nonfaulty processes.

Thus, agreement is established.

23.2 Time Dependent Algorithms

In reality, most algorithms are neither synchronous nor completely asynchronous, but somewhere in between. There is some information about the relative speeds of components, but not absolute synchrony.

Time-Dependent Algorithms arise:

- In real-time process control, "real" safety properties are important. They assume some information about the speeds of real-world and computer components, in order to guarantee some real-world safety properties, and some time bound properties. Here "safety" can be a pun - it can be a property like a train and a car don't coexist in an intersection. Examples of contexts include nuclear reactor control, and factory or airplane control.

- Many communication protocols use time information in various ways.

  - Timeouts are used to detect failure (by receiving regular "I'm alive" messages). Timeouts can also be used to detect something has happened; e.g. consider datalink problem with non-FIFO channels but an assumption that no message sits in the channel longer than real time d.

  - Suppose processes have local clocks that tell when time d has elapsed. They can use this fact and know when old messages have been purged from the channel so that they do not interfere any longer with the system. This type of consideration makes for is easy to design algorithms.

- Time-slicing access to channels can be achieved.
- Two participants may use their knowledge of time to synchronize communication.

By analogy with asynchronous and synchronous distributed computing problems, we would like to understand the capabilities of timing-based systems. What problems can they solve? How do required costs (especially time) compare to synchronous and asynchronous systems? Preliminary results have been achieved in resource allocation, synchronization, and consensus problems. Proof techniques and model ideas have been generated, but more are needed, as not all carry over from [a]synchronous problems. There is also related prior work on clock synchronization in timing-based systems. There is much more work open here; it is an interesting and important new research area.

The problems we'll discuss are variants on problems we have already studied in the asynchronous case. The difference is now we will have timing assumptions.

23.2.1 I/O Automaton as a Model

I/O automata are asynchronous. They allow arbitrary interleaving of actions. They have no timing assumptions. We used some time assumptions previously to do performance analysis on some algorithms, by imposing upper bounds of time for certain kinds of events (e.g. local step time, message delivery time, critical section time). This did not restrict the possible set of executions, nor were these assumptions used to prove correctness. More precisely, imposing only upper bounds on time for various events does not restrict the set of executions (e.g. the interleavings) that arise. It only affects the times that can be assigned to the various events.

This means that any problem (defined by a set of external action sequences) that can be solved by an I/O automaton system with added upper bounds is also solved by the same I/O automaton system without upper bounds. There is no new power here.

We need lower bounds to add power to our model.

23.2.2 Fitting Bounds into I/O automata Model

A *Timed Automaton* is a new model used by Merritt, Modeegno, and Tuttle, in a slightly more general way than we will use it here. They augment the I/O Automaton by adding upper and *lower* bounds to each class of locally controlled actions. This is natural, since we can use each class to model a separate system task.

Each IOA has a *bound map* which assigns an upper and lower bound to each class, where $0 \leq b_l(C) \leq b_u(C) \leq \infty$; $b_l \neq \infty$, and $b_u \neq 0$. The bounds are real numbers. An upper bound of ∞ means that an event need not occur, even when it is enabled; there is no need for fairness: an event occurs by some particular time. Successive actions in each class occur separated by time interval $[b_l, b_u]$.

MMT look at open intervals. They can model IOA eventual fairness. We will only look at closed intervals, for simplicity. We will abandon eventual fairness in favor of time upper bounds.

A technicality is that we can't really guarantee the designated times between occurrences of C actions - what if no C action is enabled? We will adopt the model where the time counter of a class is reset to 0 at the moment where the class is enabled. Disabling an event stops the counting. External clocking could be imposed instead. But the model obtained in this way is less general, and can be emulated with this model using dummy events.

Timed Sequences have the form $s_0, (\pi_1, t_1), s_1, (\pi_2, t_2), \ldots$. They can be finite or infinite sequences (one ends in a state if it is finite). $t_0 = 0$. Times are by convention non-decreasing: $t_0 \leq t_1 \leq \ldots$ Several events can happen at the same real time.

Zeno sequences are not allowed. A Zeno sequence is an infinite sequence that gets closer and closer to a particular time, but never reaches it. Any execution associated to such a sequence of time is not fair: it does not allow time to pass. By convention, among all infinite sequences, we only allow the ones for which times are unbounded.

A *timed execution* of A with boundmap b is represented by (A, b). Such an execution follows the steps of A correctly. That is, if we remove the time components of such an execution, we get an ordinary execution of A. The following are satisfied for each C and each i by a timed execution:

1. *Upper Bound*: Suppose $b_u(C) < \infty$. If $s_i \ni enabled(A, C)$ (a state in which an action in C is enabled), and either:

   - $i = 0$ (the execution just started)
   - $\pi_i \ni C$ (the immediately previous action was from C)
   - $s_{i-1} \ni disabled(A, C)$ (C was just enabled)

   (these are the "first start counting bounds")

   then, $\exists j > i$, with $t_j \leq t_i + b_u(C)$ s.t. either $\pi_j \ni C$ or $s_j \ni disabled(A, C)$.

2. *Lower Bound*: If $s_i \ni enabled(A, C)$, and either:

   - $i = 0$
   - $\pi_i \ni C$
   - $s_{i-1} \ni disabled(A, C)$

   then, $\nexists j > i$, with $t_j < t_i + b_l(C)$ and $\pi_j \ni C$.

This model is not the final answer. A lot of current research is involved in finding the right semantic models.

23.2.3 Timed schedules and timed behaviors

The Timed Execution definition above has safety properties (the lower and upper bounds), and a liveness property (the upper bound). The safety property of the upper bound assures that an execution can't get to a particular time without an action being taken. The lower bound ensures that an event can't occur before the lower bound time has been reached.

Sometimes, we are only interested in safety. For such cases we can use a *Timed Semi-Execution.* It is a finite timed sequence, with the conditions as above, except that condition 1. is weakened to say that, with the same premises, the possibility of an early stop of the execution is allowed: an action of class C or a disabling must occur within time $b_u(C)$ unless the execution ends prematurely:

$$t_{end} \leq t_i + b_u(C).$$

Some basic facts about semi-executions can be easily established:

- A finite timed semi-execution is a timed execution iff every locally controlled enabled action in the last state has $b_u(C) = \infty$.
- The limit of successive timed semi-executions is a timed execution if the time goes to infinity.

For example, a model of gate in a circuit may be expressed as follows. The function of a gate can be modeled by enabling its output when both of the gate inputs arrive. In Automaton A, $input_1$ and $input_2$ are either $\perp$ (undefined) or v (a value from the value set). $input_1(v)$ is an input event. $output(v)$ is enabled if inputs are non-null. Its value is a function of the inputs. The automaton has a single class, and a bound map associated with it.

The *Composition of Timed Automata* produces a new automaton which is the same as the composition of the automata, and a new bound map, which is a combination of the bound maps for all classes. A timed behavior of a composition is a composition of the timed behaviors. This *compositionality* is defined as the set of sequences of timed external actions that projects to give timed behaviors of the components.

A timed automaton *solves* a problem when its set of timed behaviors (external actions and their associated times) is a subset of the timed behaviors of the specifications. But we will not allow the timed automaton to be trivial: the empty set, which is trivially a subset of all behaviors will not be, by definition, a legal solution to the specifications.

Also, infinite compositions of timed automata are not allowed: they contradict the rule against Zeno sequences.

6.852 Distributed Algorithms **Handout 8** **September 13, 1990**

Homework Assignment Handout 8

No late homeworks accepted. *Solutions Due: September 20*

Please write up all solutions clearly, concisely, and legibly.

1. Simple I/O automata:
   (a) Define an I/O automaton representing a reliable message channel that accepts and delivers messages from the union of two alphabets, M_1 and M_2. The message channel is supposed to preserve the order of messages from the same alphabet. Also, if a message from alphabet M_1 is sent prior to another message from alphabet M_2, the corresponding deliveries must occur in the same order. However, if a message from M_1 is sent after a message from M_2, then the deliveries are permitted to occur in the opposite order.

   Specify explicitly what the action signature, states, start states, steps and partition classes are.

   (b) For your automaton, give an example of each of the following: a fair execution, a fair schedule, a fair behavior, an execution that is not fair, and a behavior that is not fair.

2. Consider the leader election algorithm in Section 4 of the CWI paper (handout 4). Prove the safety property that at most one "leader" event occurs. Hint: Formulate carefully an invariant similar to the one suggested after the code in Section 4, page 240. Prove the invariant by induction, and then show that the invariant implies the required condition.

3. Let C be the reordering channel defined in Section 4 of the CWI paper, page 239, and let A be a similar channel that delivers messages in FIFO order. (a) Give a formal definition of an I/O automaton for A. (b) Define a possibilities mapping from A to C, and show that it is actually a possibilities mapping. (The mapping should be single-valued, i.e., an abstraction mapping.)

4. Prove the following theorem:

   Let A and B be automata with the same external action signature. If there is a possibilities mapping from A to B, then $behs(A) \subseteq behs(B)$.

6.852 Distributed Algorithms **Handout 14** **September 20, 1990**

Homework Assignment Handout 14

No late homeworks accepted. *Solutions Due: September 27*

Please write up all solutions clearly, concisely, and legibly.

1. Do problem 4 from last week's assignment, which was postponed because we didn't get to the relevant material in class. That is, prove the following theorem.

   Suppose that A and B are automata with the same external action signature. If there is a possibilities mapping from A to B, then $behs(A) \subseteq behs(B)$.

2. Show that lockout is possible in Dijkstra's algorithm. That is, describe a fair execution of the algorithm that fails to grant the resource to a requesting user, (even though users always return granted resources).

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

Homework Assignment Handout 14

No late homeworks accepted. *Solutions Due: September 27*

Please write up all solutions clearly, concisely, and legibly.

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. (Open, which means we haven't got a complete answer to this one.) For Burns' algorithm, prove an upper bound (as small as you can) on the time from when any process enters the trying region until the next time *some* process (not necessarily the same process) enters the critical region. Also give a lower bound (as large as you can) for this same time, by describing a particular execution in which this time is large. Try to get the bounds as close as possible.

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

Homework Assignment Handout 23

No late homeworks accepted. *Solutions Due: October 18*

Please write up all solutions clearly, concisely, and legibly.

1. Give a direct proof of a special case of the Burns-Lynch impossibility result, saying that two 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. Design a test-and-set algorithm to solve a strong version of the 2-exclusion problem described in Programming Assignment 2. In the version we have in mind, the processes should be enabled to enter the critical region in FIFO order, based on their initial requests. However, unlike in the mutual exclusion problem, there should normally be two processes either in or enabled to enter their critical regions. (Note: this problem should be easier than the one in the programming assignment, because it allows use of powerful test-and-set shared memory.)

   Try to minimize the size of the shared variable. (Keeping a complete queue in the shared memory would work, but the variable would be quite large.) Write your algorithm using lock-unlock notation, to make the indivisible steps explicit.

3. Consider Rabin's randomized algorithm once again.

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

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

6.852 Distributed Algorithms **Handout 25** **October 18, 1990**

Homework Assignment Handout 25

No late homeworks accepted. *Solutions Due: October 25*

Please write up all solutions clearly, CONCISELY, and legibly.

1. Show how to produce a resource problem (in the form of a monotone Boolean formula, as described in class) that is equivalent to (that is, specifies the same exclusion relationships as) any given exclusion problem.
2. For Dijkstra's Dining Philosophers algorithm, describe a fair execution that locks out a particular process.
3. Give a generalized version of the Burns left-right alternating Dining Philosophers solution that works for an odd number of philosophers. For your algorithm, prove an upper bound independent of n for the maximum time a philosopher must wait to eat after becoming hungry.
4. Show that there exists an adversary for the Rabin-Lehmann randomized Dining Philosophers algorithm for which the probability of locking out a particular process is nonzero.

Homework Assignment Handout 31

No late homeworks accepted. *Solutions Due: November 8*

Please write up all solutions clearly, CONCISELY, and legibly.

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 registers are atomic, the resulting *k-ary* register is not atomic.

2. (Leader election)

   2.1. Give the best upper bound you can on the time complexity for Peterson's ring leader election algorithm.

   2.2. Write a bidirectional version of Peterson's leader election algorithm. What are the message and time complexities of your algorithm?

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. 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 2?

5. Do a careful sketch of a possibilities mapping from Bloom's Boolean-tag 2-writer algorithm to the integer-tag version given in class.

6.852 Distributed Algorithms **Handout 36** **November 8, 1990**

Homework Assignment Handout 36

No late homeworks accepted. *Solutions Due: November 15*

Please write up all solutions clearly, CONCISELY, and legibly.

1. Consider the Loui, Abu-Amara impossibility result for wait-free consensus in a read-write shared memory model. Show that this impossibility result still holds in the case where the algorithm is required to tolerate only a single process failure (rather than the arbitrary number of failures required for wait-free algorithms). (If you get stuck, you may find it useful to look at some of the relevant research papers.)

2. Design a new distributed Minimum Spanning Tree algorithm. (You may describe it in words instead of code, if you like.) This new algorithm should be a lot simpler than the Gallager et al algorithm. It should solve the same problem, however, and with time complexity that is approximately as good (that is, $O(nlogn)$). However, it need not be so highly optimized for the number of messages. (Hint: Your algorithm could again be based on combining fragments, where each fragment has an associated level number. Again, each individual node can start as a single-node fragment of level 0. This time, fragments of level l can only combine to give fragments of level $l+1$. Nodes can be synchronized to operate in "phases" corresponding to fragment levels.)

3. 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.

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 that 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.)

Homework Assignment Handout 38

No late homeworks accepted. *Solutions Due: November 27*

Please write up all solutions clearly, CONCISELY, and legibly.

1. Write the Chandy-Misra dining philosophers algorithm and the Chandy-Misra drinking philosophers algorithm as I/O automata (Using precondition-effect notation).

2. Try to design a drinking philosophers algorithm with good time complexity. State your time bound claims carefully. Can you get the time complexity to depend on the actual requests rather then the potential requests? (Note: We haven't worked this one out.)

3. We have seen in class that distributed consensus is impossible in the presence of stopping failures, for deterministic asynchronous algorithms. However, an approximate version of the consensus problem is solvable in such a setting.

   In the approximate problem we have in mind, each node receives an $\text{init}_i(v)$ input for some real number v, and is supposed to eventually perform a $\text{dec}_i(w)$ output, for a real number w. There are two requirements:

   3.1. *agreement*: all decisions are within some predetermined ϵ of each other,

   3.2. *validity*: all decisions are in the range of the inputs.

   Design an algorithm for this approximate agreement problem. It should tolerate a predetermined number f of faults. In order to do this, you will need to assume that the total number of processes is sufficiently large compared to f. (How large?)

   You may allow messages to contain real numbers.

6.852 Distributed Algorithms **Handout 43** **November 29, 1990**

Homework Assignment Handout 43

No late homeworks accepted. *Solutions Due: December 6*

Please write up all solutions clearly, CONCISELY, and legibly.

1. Design variants of BenOr's randomized consensus protocol that work for the following two cases. In each case, try to design the algorithm to use fewer than the 7t+1 processes used in the asynchronous, Byzantine case. (a) Asynchronous system with stopping faults rather than Byzantine faults. (b) Synchronous system with Byzantine faults.

2. Give a detailed description (i.e., as I/O automata) of the "filter" processes that implement the Chandy-Lamport global snapshot algorithm.

3. Handouts 40 and 41 contain a description and proof of a data link protocol that guarantees reliable message delivery using bit headers and non-FIFO channels. Describe an execution $\alpha = \beta\gamma$ and a function f with the following properties:

   3.1. β includes exactly one *send-msg* event and exactly one (corresponding) *rcv-msg* event, and also contains exactly l lost packets from the transmitter to the receiver, and exactly l lost packets from the receiver to the transmitter, for a given constant l.

   3.2. The number of packets used to deliver the k^{th} message in α is at most $f(k)$.

   3.3. f is as small as you can obtain.

4. Design a variant of the protocol presented in class for agreement in the presence of stopping faults in synchronous systems; your variant should use only a polynomial number of messages in the worst case rather than the exponential number used by the algorithm described in class.

5. (Optional) Discuss the usefulness of Lamport's state machine approach for the distributed solution of any of the problems considered in this course.

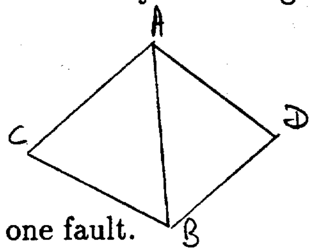
6.852 Distributed Algorithms	**Handout 45**	**December 7, 1990**

Homework Assignment Handout 45

No late homeworks accepted. *Solutions Due: December 13*

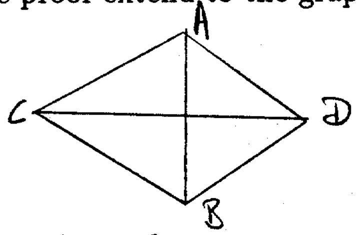
THIS IS THE LAST HOMEWORK ASSIGNMENT!
Please write up all solutions clearly, CONCISELY, and legibly.

1. 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:



2. Design a randomized asynchronous agreement algorithm for agreement on an arbitrary value set V rather than just $\{0, 1\}$. Hint: Combine the ideas of Turpin and Coan with those of BenOr. How many processes are required?

3. The proof given in class of the lower bound of $f + 1$ on the number of synchronous rounds required for consensus is based on the construction of a chain of executions, starting from one in which the decision must be 0 and ending with one in which the decision must be 1. How long is the chain (in terms of n and f)?

4. Prove that for any undirected graph, there exists a spanning forest with trees of logarithmic size, such that the number of neighboring tree pairs (i.e. there exists an edge with endpoints belonging to those trees) is linear in the number of nodes.

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] Karl Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the 7^{th} Annual ACM Symposium on Principles of Distributed Computing*, pages 291–302, Toronto, Canada, August 1988.

[4] 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.

[5] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic snapshots of shared memory. In *Proceedings of the 9^{th} Annual ACM Symposium on Principles of Distributed Computing*, Quebec, Canada, August 1990. Also, Technical Memo MIT/LCS/TM-429, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May 1990. Submitted to *Journal of the ACM*.

[6] 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.

[7] Yehuda Afek, Hagit Attiya, Alan Fekete, Nancy Lynch, Yishay Mansour, Da-Wei Wang, and Lenore Zuck. Reliable communication over unreliable channels. Manuscript.

[8] Yehuda Afek and Eli Gafni. Bootstrap network resynchronization: An effecient technique for end-to end communication, 1990.

[9] Yehuda Afek, Eli Gafni, and Adi Rosen. Slide - a technique for communication in unreliable networks (extended abstract), 1990.

[10] 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.

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

[12] E. Arjomandi, M. Fischer, and N. Lynch. A difference in efficiency between synchronous and asynchronous systems. Technical Report GIT-ICS-81/07, Georgia Institute of Technology, June 1981.

[13] E. Arjomandi, M. Fischer, and N. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the ACM*, 30(3):449–456, July 1983.

[14] 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.

[15] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischu. Renaming in an asynchronous environment. *J. ACM*, 37(3), July 1990.

[16] H. Attiya, C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty.

[17] H. Attiya and N. Lynch. Time bounds for real-time process control in the presence of timing uncertainty. In *Proceedings of the* 10th *IEEE Real-Time Systems Symposium*, Santa-Monica, December 1989. Expanded version: Technical Memo MIT/LCS/TM-403, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1989. Submitted for publication.

[18] H. Attiya, N. Lynch, and N. Shavit. Are wait-free algorithms fast?

[19] H. Attiya, M. Snir, and M. Warmuth. Computing in an anonymous ring. *Journal of the ACM*, 35(4):845–876, October 1988.

[20] Hagit Attiya and Marios Mavronicolas. Efficiency of asynchronous vs. semi-synchronous networks. Submitted to *the 28th annual Allerton Conference on Communication, Control and Computing*, 1990.

[21] 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.

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

[23] 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.

[24] 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.

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

[26] Baruch Awerbuch, Oded Goldreich, David Peleg, and Ronen Vainish. A tradeoff between information and communication in broadcast protocols, 1990. To appear in JACM.

[27] 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.

[28] Amotz Bar-Noy, Danny Dolev, Cynthia Dwork, and Raymond Strong. Shifting gears: Changing algorithms on the fly to expedite Byzantine agreement, June 27 1990.

[29] 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.

[30] Piotr Berman and Juan Garay. Cloture voting: n/4-resilient distributed consensus in t+1 rounds, 1990.

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

[32] 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.

[33] 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.*

[34] G. Bracha. An $o(\log n)$ expected rounds randomized *byzantine* generals algorithm. In *Proceedings of 17th Symposium on Theory of Computing*, pages 316–326, May 1985. Journal of ACM, 34(4):910-920,1987.

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

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] James Burns, Mohamed Gouda, and Raymond Miller. Stabilization and pseudo-stabilization. Technical report TR-90-13, University of Texas at Austin, May 1990.

[42] James Burns and Jan Pachl. Uniform self-stabilizing rings. *Journal of the ACM*, 11(2):330–344, April 1989.

[43] James E. Burns and Nancy A. Lynch. Bounds on shared memory for mutual exclusion, 1990.

[44] O. Carvalho and G. Roucairol. Assertion, decomposition and partial correctness of distributed control algorithms. *Distributed Computing Systems*, pages 67–93, 1983.

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

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

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

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

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

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

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

[52] 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.

[53] Soma Chaudhuri and Jennifer Welch. Bounds on the costs of register implementations. Technical Report TR90-025, University of North Carolina at Chapel Hill, June 1990.

[54] B. Chor and B. Coan. A simple and efficient randomized Byzantine agreement algorithm. In *IEEE Transactions on Software Engineering*, volume SE-11, 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.

[55] Benny Chor and Cynthia Dwork. Randomization in Byzantine agreement. *Advances in Computing Research*, 5:443–497, 1989.

[56] 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.

[57] 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.

[58] 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.

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

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

[61] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communication of the ACM*, 17(11):643–644, November 1974.

[62] E.W. Dijkstra. Solution of a problem in concurrent programming control. *Communications Of The ACM*, 8(9):569, September 1965.

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

[64] D. Dolev. The Byzantine generals strike again. *Journal of Algorithms*, 3:14–30, 1982.

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

[66] 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.

[67] D. 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.

[68] 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.

[69] 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.

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

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

[72] 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.*

[73] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 1988. To appear.

[74] 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.

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

[76] 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.

[77] A. Fekete and N. Lynch. The need for headers: an impossibility result for communication over unreliable channels. Also, Technical Memo MIT/LCS/TM-428, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, May, 1990. Submitted for publication. To appear in CONCUR. 1990.

[78] A. Fekete, N. Lynch, Y. Mansour, and J Spinelli. The data link layer: The impossibility of implementation in face of crashes. Technical Memo MIT/LCS/TM-355.b, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1989. Submitted for publication.

[79] 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.

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

[81] M. Fischer, N. Griffeth, and N. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982. Also, in *Proceedings of IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, PA, July 1981, 33-38.

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

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

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

October, 1985. Also, ACM Transactions on Programming Languages and Systems, Vol 11, No. 1, January 1989, pages 90-114.

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

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

[87] M. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. Research Report 221, Yale University, February 1982.

[88] G. 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.

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

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

[91] 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, MIT/LCS/TR-390, MIT Laboratory for Computer Science, Cambridge, MA, May 1987.

[92] Kenneth Goldman and Nancy Lynch. Modelling shared state in a shared action model. In *Proceedings 5th Annual IEEE Symposium on Logic in Computer Science*, pages 450–463, Philadephia, PA., June 1990.

[93] Mohamed Gouda and Nicholas Multari. Stabilizing communication protocols. Technical report TR-90-20, University of Texas at Austin, June 1990.

[94] 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.).

[95] 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.

[96] Joseph Y. Halpern and Lenore D. Zuck. A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols. In *Proceedings of the* 6th *Annual ACM Symposium on Principles of Distributed Computing*, pages 269–280, August 1987.

[97] 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.

[98] 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.*

[99] 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.

[100] Amir Herzberg and Shay Kutten. Fast isolation of arbitrary forwarding-faults, 1989.

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

[102] 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.

[103] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1984.

[104] 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.

[105] Shmuel Katz and Kenneth J. Perry. Self-stabilizing extensions for message-passing systems. In *Proceedings of the* 9th *Annual ACM Symposium on Principles of Distributed Computing*, pages 91–101, Quebec, Canada, August 1990.

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

[107] 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.

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

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

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

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

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

[113] L. Lamport. On interprocess communication. *Distributed Computing*, 1(1):77–85, 86–101, 1986. *Digital Systems Research* TM-8.

[114] L. Lamport and N. Lynch. Chapter on distributed computing. *Handbook of Theoretical Computer Science.*

[115] L. Lamport and N. Lynch. Distributed computing. Chapter of Handbook on Theoretical Computer Science. Also, appeared as Technical Memo MIT/LCS/TM-384, Laboratoryfor Computer Science, Massachusetts Institute of Technology, Cambridge, MA, February 1989.

[116] 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.

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

[118] Leslie Lamport. The part-time parliament. Technical Memo 49, Digital Systems Research Center, September 1 1989.

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

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

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

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

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

[124] 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.

[125] 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.

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

[127] 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.

[128] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. Atomic transactions. Book in progress.

[129] 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.

[130] 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.

[131] Nancy Lynch and Hagit Attiya. Using mappings to prove timing properties. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, Quebec, Canada, August 1990. Expanded version: Technical Memo MIT/LCS/TM-412.b, Laboratory for Computer Science, Massachusetts Institute of Technology, December 1989. Submitted for publication.

[132] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, August 1987. A full version is available as MIT Technical Report MIT/LCS/TR–387.

[133] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3), 1989.

[134] 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.

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

[136] Michael Merritt, Francesmary Modugno, and Mark Tuttle. Time constrained automata, July 23 1990.

[137] 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.

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

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

[140] 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.

[141] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance for distributed algorithms. Technical Report TR90-1081, Cornell University, January 1990.

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

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

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

[145] 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.

[146] 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.

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

[148] 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.

[149] 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.

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

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

[152] 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).

[153] I. Saias and N. Lynch. An analysis of Rabin's randomized mutual exclusion algorithm. MIT/LCS/TM-462, December 1991.

[154] 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.

[155] 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.

[156] Russel Schaffer and Bard Bloom. On the correctness of atomic multi-writer registers. Technical Memo MIT/LCS/TM–364, MIT Laboratory for Computer Science, June 1988.

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

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

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

[160] Ray Strong, Danny Dolev, and Flaviu Cristian. New latency bounds for atomic broadcast, April 1990.

[161] Ewan Tempero and Richard Ladner. Tight bounds for weakly bounded protocols. In *Proceedings of the* 9^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 205–212, Quebec, Canada, August 1990.

[162] 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.

[163] M. Tuttle. *Knowlege and Distributed Computation.* PhD thesis, MIT Electrical Engineering and Computer Science, September 1989. Also, Technical Report MIT/LCS/TR-477, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA. 1990. Supervised by Nancy Lynch.

[164] 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.

[165] Da-Wei Wang and Lenore Zuck. Tight bounds for the sequence transmission problem. In *Proceedings of the* 8^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 73–83, August 1989.

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

Index