Global States of a Distributed System

MICHAEL J. FISCHER, NANCY D. GRIFFETH, AND NANCY A. LYNCH

Abstract—**A global state of a distributed transaction system is *consistent* if no transactions are in progress. A *global checkpoint* is a transaction which must view a globally consistent system state for correct operation. We present an algorithm for adding global checkpoint transactions to an arbitrary distributed transaction system. The algorithm is nonintrusive in the sense that checkpoint transactions do not interfere with ordinary transactions in progress; however, the checkpoint transactions still produce meaningful results.**

Index Terms—**Checkpoint, consistency, distributed system, global state, transaction.**

I. Introduction

COMPUTING systems operate by a sequence of internal transitions on the global state of the system. The global state represents the collective state of a set of *objects* which the system controls. Often many primitive state transitions are necessary to accomplish a larger semantically meaningful task, called a *transaction*. Transactions are designed to take the system from one meaningful or *consistent* state to another, but during the execution of the transaction, the system may go through inconsistent intermediate states. Thus, to insure consistency of the system state, every transaction must either be run to completion or not run at all.

Transactions are often the basis for concurrency control. In a distributed database system, a standard criterion for correctness of a system is that all allowable interleavings of transactions be "serializable" (cf. [1]). However, there are systems which can run acceptably with unconstrained interleavings. In a banking system, for example, a transfer transaction might consist of a withdrawal step followed by a deposit step. In order to obtain fast performance, the withdrawals and deposits of different transfers might be allowed to interleave arbitrarily, even though the users of the banking system are thereby presented with a view of the account balances which includes the possibility of money being "in transit" from one account to another.

One useful kind of transaction is a "checkpoint"—a transaction that reads and returns all the current values for the objects of the system. In a bank database, a checkpoint can be used to audit all of the account balances (or the sum of all account balances). In a population database, a checkpoint can be used to produce a census. In a general transaction system, the checkpoint can be used for failure detection and recovery: if a checkpoint produces an inconsistent system state, one assumes that an error has occurred and takes appropriate recovery measures.

For a checkpoint transaction to return a meaningful result, the individual read steps of the checkpoint must not be permitted to interleave with the steps of the other transactions; otherwise an inconsistent state can be returned even for a correctly operating system, and it might be quite difficult to obtain useful information from such intermediate results. For example, in a bank database with transfer operations, an arbitrarily interleaved audit might completely miss counting some money in transit or count some transferred money twice, thereby arriving at an incorrect value for the sum of all the account balances.

A checkpoint which is not allowed to interleave with any other transactions is called a *global checkpoint*. In the bank database, a global checkpoint would only see completed transfers; no money would be overlooked in transit, and a correct sum would be obtained for all account balances. In general, a global checkpoint views a *globally consistent state* of the system.

In this paper, we present a method of implementing global checkpoints in general distributed transaction systems. We assume one starts with an underlying distributed transaction system known to be correct. Next we add some checkpoint transactions C which are known to be correct if run when no other transactions are running. Call the resulting system S. Finally, we show how to transform S into a new system S' which does the "same" thing as S and which turns each of the transactions in C into a global checkpoint, i.e., one that always returns a view of a globally consistent system state of the underlying transaction system.

Our introduction of the global checkpoints is "nonintrusive" in the sense that no operations of the underlying system need to be halted while the global checkpoint is being executed. Because of this, it is not always possible to have the global checkpoint view a consistent state in the recent history of the underlying transaction system, for that system might enter consistent states only infrequently because of heavy transaction traffic. Thus, instead of viewing a consistent state that *actually* occurs, our global checkpoints view a state that *could* result by running to completion all of the transactions that are in progress when the global checkpoint begins, as well as some of the transactions that are initiated during its execution.

II. A Model for Asynchronous Parallel Processes

The formal model used to state the correctness conditions and describe the algorithm is that of [2]. Only a brief descrip-

Manuscript received February 15, 1981. This work was supported in part by the National Science Foundation under Grants MCS77-02474, MCS77-15628, MCS80-03337, and MCS79-24370, the U.S. Army Research Office under Contract DAAG29-79-C-0155, and the Office of Naval Research under Contracts N00014-79-C-0873 and N00014-80-C-0221. An earlier version of this paper was presented at the IEEE Symposium on Reliability in Distributed Software and Database Systems, Pittsburgh, PA, July 21–22, 1981.

M. J. Fischer is with the Department of Computer Science, Yale University, New Haven, CT 06520.

N. D. Griffeth and N. A. Lynch are with the School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332.

0098-5589/82/0500-0198$00.75

tion is provided in this paper; the reader is referred to [2] for a complete, rigorous treatment.

The basic entities of the model are *processes* (automata) and *variables*. Processes have *states* (including start states and possibly also final states), while variables take on *values*. An atomic *execution step* of a process involves accessing one variable and possibly changing the process' state or the variable's value or both. A *system of processes* is a set of processes, with certain of its variables designated as *internal* and others as *external*. Internal variables are to be used only by the given system. External variables are assumed to be accessible to some "environment" (e.g., other processes or users) which can change the values between steps of the given system.

The execution of a system of processes is described by a set of *execution sequences*. Each sequence is a (finite or infinite) list of steps which the system could perform when interleaved with appropriate actions by the environment. Each sequence is obtained by interleaving sequences of steps of the processes of the system. Each process must have infinitely many steps in the sequence unless that process reaches a final state.

For describing the external behavior of a system, certain information in the execution sequences is irrelevant. The *external behavior* of a system S of processes, extbeh(S), is the set of sequences derived from the execution sequences by "erasing" information about process identity, changes of process state and accesses to internal variables. What remains is just the history of accesses to external variables. This history takes the form of a sequence of *variable actions*, which are triples of the form (u, X, v), where u is the old value read from the variable X, and v is the new value written by an atomic step of the system. The external behavior completely characterizes the system from the user's point of view; two systems with the same external behavior are completely indistinguishable to the user.

III. An Abstract Distributed Transaction System

In a database system, a transaction is usually considered to be a sequence of operations on the database entities which should be performed according to some concurrency control policy. For our purposes, we do not need to look inside the transactions—all that we require is that a particular transaction can be requested at any time, and once requested, it will eventually run to completion. What the transaction does while it is running and how it interacts with other concurrent transactions does not concern us. We simply assume a distributed system which understands the initiation and completion of transactions at its external variables.

We make the technical restriction that each transaction can be invoked only once; thus, our transactions should be thought of as instances of the usual database notion of transaction. We also assume that an infinite number of transactions are possible, although only a finite number can be running at any given time; thus, our systems never stop.

Formally, an *abstract transaction system* is a distributed system whose external variables, called *ports*, have a special interpretation. Let T be an infinite set of *transactions*. Each port can contain a finite set of *transaction status words*, each of which is a triple (t, a, s), where $t \in T$, a is an arbitrary parameter or result value of the transactions, and $s \in$ {'RUNNING', 'COMPLETE'} describes the state of the transaction. We require that each port can be accessed by only one process, called the *owner* of the port.

The intended operation of the system is as follows. A user initiates a transaction t with argument a by inserting the triple $(t, a,$ 'RUNNING') into the set of transaction status words in some port. Eventually, the system replaces that triple by a new triple $(t, b,$ 'COMPLETE') in the same port. The value b is the *result* of the transaction. We assume the user behaves correctly in not trying to initiate the same transaction more than once and in not modifying the transaction status word once a transaction has been initiated. Likewise, a correct abstract transaction system never changes the ports or modifies the transaction status words except as described above.

Thus, a correct abstract transaction system running with a correct user maintains a global invariant that for each transaction $t \in T$, there is at most one port containing a transaction status word with t as first component, and there is at most one such word in that port. We call that word, if it exists, the *status word* for t, and we say t is *running* or *completed* depending on the third component of its status word. We say t is *latent* if it has no status word. The conditions above imply that the only possible transitions in the status of a transaction are from latent to running and from running to completed; moreover, every running transaction eventually becomes completed. Note also that there is no *a priori* bound on the number of transactions that can be running simultaneously.

IV. Checkpoint Transactions

Let $C \subseteq T$ be a distinguished set of transactions called *checkpoints*. Members of $T - C$ are called *ordinary transactions*. The execution of a checkpoint transaction and the result it returns are *valid* if no other transaction is running while the checkpoint is. While we make no restrictions on what a checkpoint does, the intuition is that a checkpoint needs to look at a globally consistent system state in order to work properly, that is, a state of the system that occurs when no transactions are in progress. For example, the checkpoint might be an audit of the account balances in a simple banking system, or it might be a consistency check in a file system. These two examples are pursued further in Section VI.

Our goal in this paper is, given a transaction system S with checkpoints, to construct a new transaction system S' which does the "same" thing as S for noncheckpoint transactions and which returns a valid result for each checkpoint transaction. A straightforward implementation of S' would simply suspend further processing of transactions when a checkpoint is requested, wait for any transactions currently in progress to complete, and then run the checkpoint. After the checkpoint had been completed, normal processing of transactions could be resumed.

In many practical situations, however, such a solution is highly undesirable, for the entire system must wait while a checkpoint is being performed. This is likely to take a considerable length of time since checkpoints may require reading the entire system state.

In Section V, we present a solution which permits checkpoints to be run concurrently with the normal processing of ordinary transactions. The price we pay is in having a slightly less appealing correctness condition for the result of the check-

point. Since normal transactions are not suspended, the system may never reach a globally consistent state, so it is not obvious how a meaningful result can be obtained at all. Our approach is to run the checkpoint on a globally consistent state obtained by the following steps.

Step 1: Disable the initiation of further transactions at each of the ports.

Step 2: Run to completion any transactions in progress.

To do this and still not interfere with the processing of normal transactions requires that we split the computation into two parallel branches. One branch continues to simulate S on the ordinary transactions; the other branch handles the execution of the checkpoint as described above. When the checkpoint is complete, the result is stored back in the appropriate transaction status word and the branch discarded.

Consequences of this strategy are as follows.

1) The value returned by a checkpoint does not reflect what *actually* happened in the history of execution; only what might have happened if certain transactions initiated after the start of the checkpoint had not occurred.

2) Any side effects of checkpoint transactions are discarded, so other transactions continue to operate as if no checkpoints had ever taken place.

With this motivation in mind, we now turn to the definitions needed to state the formal correctness conditions for the system S'.

Let X be a port and u, v be sets of transaction status words. We call the variable action (u, X, v) a *port action*. Let PA be the set of all port actions. A *behavior sequence* is a finite or infinite sequence of port actions, i.e., a member of $\mathfrak{B} = PA^* \cup PA^\infty$.

Let h be a map which erases checkpoint status words from port values, that is, if u is a set of transaction status words, then

$$h(u) = \{(t, a, s) \mid (t, a, s) \in u \text{ and } t \in T - C\}.$$

Extend h to port actions by

$$h((u, X, v)) = (h(u), X, h(v)).$$

Extend h further to $\mathfrak{B}$ by applying it componentwise.

Let $e \in \mathfrak{B}$. Define two functions:

$$\text{running}(e) = \{t \in T \mid e = e_1 \cdot (u, X, v) \cdot e_2 \text{ and } (t, a, \text{'RUNNING'}) \in u \text{ for } (t, a, \text{'RUNNING'}) \text{ some transaction status word, } e_1 \in PA^*, (u, X, v) \in PA, \text{ and } e_2 \in \mathfrak{B}\}.$$

$$\text{completed}(e) = \{t \in T \mid e = e_1 \cdot (u, X, v) \cdot e_2 \text{ and } (t, b, \text{'COMPLETED'}) \in v \text{ for } (t, b, \text{'COMPLETED'}) \text{ some transaction status word, } e_1 \in PA^*, (u, X, v) \in PA, \text{ and } e_2 \in \mathfrak{B}\}.$$

Thus, running(e) is the set of transactions which are running at some time during e, and completed(e) is the set of transactions which have completed in e.

Let $e \in \mathfrak{B}$, $t \in \text{running}(e)$, and $i \in N$. *t starts at step i of e* if i is the length of the longest prefix e_1 of e for which $t \notin \text{running}(e_1)$.

An abstract distributed transaction system S' is a *faithful implementation* of a system S with checkpoint set C if the following conditions hold.

1) Faithfulness: Let $e \in \text{extbeh}(S)$ such that $h(e) = e$ (i.e., e contains no checkpoint transactions). Let σ: $C \to N$ be a partial function with domain dom(σ). Then there exists $e' \in \text{extbeh}(S')$ such that $h(e') = e$, $\text{running}(e') = \text{running}(e) \cup \text{dom}(\sigma)$, and for all $t \in \text{dom}(\sigma)$, t starts at step $\sigma(i)$ in e'.

2) Safety: Let $e' \in \text{extbeh}(S')$. Then $h(e') \in \text{extbeh}(S)$.

3) Validity of Checkpoints: Let $e' \in \text{extbeh}(S')$, $c \in C$, and suppose c runs to completion in e' and produces result b. Let i be the step at which c starts in e', and let e'_1 be the prefix of e' of length i. Let e'_2 be the shortest word such that $e'_1 e'_2$ is a prefix of e' and $c \in \text{completed}(e'_1 e'_2)$. Then there exists $e \in \text{extbeh}(S)$ such that c runs to completion in e and produces result b, and e satisfies the following. There exist words e_1, e_2, f such that $e_1 e_2 f$ is a prefix of e, and

a) $h(e_1 e_2) = e_1 e_2$;

b) $h(e'_1) = e_1$;

c) $\text{running}(e_1 e_2) \subseteq \text{running}(e'_1 e'_2)$;

d) $c \in \text{completed}(e_1 e_2 f)$ and $\{c\} = \text{running}(e_1 e_2 f) - \text{completed}(e_1 e_2)$.

Conditions 1) and 2) insure that S' faithfully simulates S on the noncheckpoint transactions and that the presence or absence of checkpoint transactions does not affect the processing of other transactions by S'. Condition 3) insures that S' computes acceptable results for the checkpoint transactions. In particular, the result of each checkpoint must be a value obtainable by some computation of S which i) runs no checkpoints before the given checkpoint, ii) agrees with the computation of S' up to the point where the checkpoint began (again ignoring other checkpoints), iii) only initiates transactions thereafter which actually occurred in S', and iv) runs the checkpoint after all the transactions in progress at the time of the checkpoint request together with any transactions initiated after the checkpoint have completed, thereby insuring a valid result.

V. A Faithful Implementation

Given an abstract distributed transaction system S with checkpoint set C, we sketch how to construct a new system S' which faithfully implements S.

S' operates by simulating a number of copies of S: a "base" copy S_0 and a copy S_c for each $c \in C$. S_0 processes all of the noncheckpoint transaction requests received by S', and S_c processes checkpoint transaction c.

S_0 ignores checkpoints but otherwise acts just like S. S_c, $c \in C$, does exactly the same thing as S_0 up until checkpoint c is requested. At that time, the computation of S_c begins to diverge from that of S_0. S_c continues behaving like S, but it starts ignoring certain new transactions that are being processed by S_0. Eventually, it ceases processing new transactions entirely, and all the transactions currently in progress are run to completion. At that time, S_c runs checkpoint transaction c, and when it completes, S_c writes the result back into the transaction status word at the initiating port. S_c has then completed its task and can terminate.

The structure of S' is similar to that of S. Each process and variable of S has a corresponding process or variable in S'. Process k of S' simulates process k in each of the S_i, $i \in \{0\} \cup C$. Similarly, internal variable X of S' simulates internal variable X in each of the S_i. The states of processes in S' are labeled sets of states of corresponding processes of S, and values of variables in S' are labeled sets of values of corresponding variables of S, where the labels are taken from $\{0\} \cup C$. S and S' have identical ports and port values.

We now describe in some detail the operation of the processes in S_c. Each process does exactly the same thing as the corresponding process of S_0 until it learns that checkpoint c has been requested. There are three ways that a process might learn this. It might access its port and see the transaction status word for c. In this case, that process is called the *checkpoint initiator*. Secondly, it might receive a "message" from the checkpoint initiator informing it of the start of the checkpoint. Finally, it might read an internal variable and detect that the computation of S_c has begun to diverge from that of S_0, enabling it to deduce that the checkpoint has started.

When the checkpoint initiator discovers the start of the checkpoint, it broadcasts this fact to the other processes of S_c. Each process of S_c upon learning of the initiation of the checkpoint makes a private copy of its port and thereafter refers to its private copy rather than the real port. In this way, future transaction requests are ignored by S_c, and results of transactions produced by S_c (which might differ from those produced by S_0) do not affect the real ports. When a process of S_c finally discovers that all of the transactions at its port have completed, it sends back an acknowledgment to the checkpoint initiator. When the initiator has received an acknowledgment from each process (including itself), it begins processing the checkpoint by placing the checkpoint request in its own private copy of its port. All of the processes of S_c continue operating and serve collectively to process the checkpoint c. When c completes, the checkpoint initiator copies the final transaction status word for c from the private copy of its port back into the real port.

The correctness conditions of Section IV are quite strong and do not permit S' to make any accesses to the ports other than those made by S_0. Therefore, the simulation of the S_c, $c \in C$, must be coordinated with that of S_0 so that all real port accesses by S_c are "piggybacked" onto port access by S_0. The basic strategy is that S_0 runs freely, but a process of S_c wishing to access the real port must wait until the corresponding process of S_0 is ready to make its next port access. The two (or more) accesses are then combined into one and performed simultaneously. The accesses never conflict because each process of S_c does the exact same thing as the corresponding process of S_0 up until the point where it discovers the start of the checkpoint. Thereafter, it only modifies the status word for c, whereas processes of S_0 only modify status words for ordinary transactions.

At any point in the computation, only a finite set D of checkpoints have ever been initiated, so the computation of every S_c, $c \in C - D$, is identical to the computation of S_0 and need be represented only once. As soon as a process of S' discovers that checkpoint c is in progress, either by being the checkpoint initiator, receiving a message from the checkpoint initiator, or by reading an internal variable in which the cth component differs from the 0th, it splits the simulation of S_c from that of S_0 and from then on, the two simulations continue independently, as described above. Hence, S' actually simulates a finite but growing set of computations.

In order to carry out the above implementation, S' needs a mechanism which permits the checkpoint initiator to communicate with every other process. In any particular application, such a communication mechanism would probably already exist in the underlying system S. However, if it is not already there, then we require that S' be augmented with such a facility.

Theorem: Let S be an abstract distributed transaction system with checkpoint set C, and let S' be the system described above. Then S' is a faithful implementation of S.

Proof Sketch: We omit the tedious but straightforward verification that S' satisfies the conditions for being a faithful implementation of S. It remains to verify however that S' is a correct abstract distributed transaction system, that is, that every transaction which is requested will eventually run to completion.

This property holds for noncheckpoint transactions by the safety property and the fact that it holds for S. It holds for checkpoint transactions because each of the phases in processing a checkpoint terminates. Eventually a request for checkpoint c gets noticed by the process of S_c which owns the port; otherwise, S_c and hence S would fail to process future transactions originating at that port. After the checkpoint request is noticed, the checkpoint initiator notifies all other processes of S_c; hence, eventually all of the other processes learn of the request. After each process becomes aware of the checkpoint, it stops accepting requests for new transactions; hence, eventually S_c stops processing new transactions. S_c continues to simulate S on the transactions that it has accepted; they all eventually complete since they would in S. Each process eventually acknowledges completion to the initiator, so eventually the checkpoint transaction itself is started. S_c continues to simulate S, so eventually the checkpoint transaction will complete and produce a valid result, which is copied back into the port.

Hence, S' is an abstract distributed transaction system which faithfully implements S, as required. □

We remark that under certain naturally occurring conditions, the efficiency of S' can be made to approach the efficiency of S. Namely, assume that all checkpoint transactions originate at the same port. Then it is an easy matter to modify the checkpoint initiator so that only one checkpoint is handled at a time. If several are requested simultaneously, the initiator will pick one to process and wait until it completes before handling another. Since only one checkpoint c is running at a time, each process of S' need only simulate two processes: the corresponding process of S_0 and the corresponding process of S_c. When a process becomes aware of the request of some checkpoint S_d, $d \neq c$, then it knows that checkpoint c must have completed; hence it terminates the simulation of S_c. Thus, the storage needed by S' for the internal variables and process states is only double that of S. (In practice, one would probably only keep duplicate copies of those objects for which the two executions S_0, S_c really produce different values.) Likewise, the time required by S', when appropriately measured, should be at worst double that of S on the particular computations actually simulated.

VI. Applications of Global Checkpoints

Global checkpoints can play an important role in the design of distributed systems for error detection, error recovery, or both.

For error detection, their use is in identifying inconsistencies in global system states that should be consistent. We have already alluded to this use in the simple banking system example in which the only allowable transactions are to transfer funds from one account to another. The sum of the account balances is the same in every globally consistent state. Therefore, our algorithm can be used to obtain that sum by running a global checkpoint transaction which simply reads each of the account balances and adds them all up. An error is indicated if this sum is not what was expected.

A similar situation occurs in the design of file systems. Often a directory must be kept consistent with the actual contents of a disk. A global checkpoint might read the items in the directory and check that they correspond with what is really on the disk. As long as no directory modification transactions were in progress when the checking was done, then a discrepancy would indicate a true file system error. Our global checkpoint algorithm can be used to detect such inconsistencies.

For error recovery, global checkpoints can be used to save the relevant part of the global state of the system so that in the event of a crash, the system can later be restarted from that point in the computation. For example, a global checkpoint could be used to provide a restart capability in the migrating transaction model of Rosenkrantz *et al.* [4] by having it return the values of all of the entities in the database.

Another such application arises in the use of the Eden system which is being developed at the University of Washington [3]. That system is object-based and includes as a primitive kernel operation a checkpoint operation that writes a single object to stable storage. The object itself decides when it is in a consistent state and hence when the checkpoint can be performed. If the object later crashes, it is restored from the version on stable storage. To extend this checkpoint facility to groups of related and cooperating Eden objects requires that the objects coordinate their checkpointing activities so that the versions saved on stable storage are globally and not just locally consistent. That is just the problem we have been treating in this paper if we take "transaction" to mean the portion of computation that an individual Eden object is in an inconsistent state, and if we assume further that an object only enters an inconsistent state in response to some external stimulus (corresponding to a transaction request). Our global checkpoint algorithm could then be applied to produce a globally consistent system state on stable storage by running the "global checkpoint" transaction which simply checkpoints each of the objects in the group. Note that our algorithm requires the independence of transactions. If one transaction can initiate another and then wait for its completion, then completion of the first depends on completion of the second, and our algorithm, which might decide to exclude the second from a checkpoint, would wait forever for the first transaction to complete. Our formal definition of a transaction system excludes the possibility of system-initiated transactions.

Acknowledgment

The authors would like to thank Mr. M. Merritt for many helpful comments and suggestions.

References

[1] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 624–633, Nov. 1976.

[2] N. A. Lynch and M. J. Fischer, "On describing the behavior and implementation of distributed systems," *Theor. Comput. Sci.*, vol. 13, pp. 17–43, 1981.

[3] E. D. Lazowska, H. M. Levy, G. T. Almes, M. J. Fischer, R. J. Fowler, and S. C. Vestal, "The architecture of the Eden system," in *Proc. 8th Symp. Operating Syst. Principles*, Dec. 1981, pp. 148–159, ACM Order 534810.

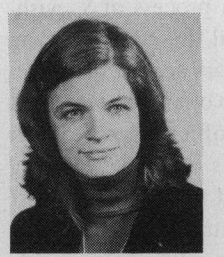
[4] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, "System level concurrency control for distributed database systems," *ACM Trans. Database Syst.*, vol. 3, pp. 178–198, June 1978.

Michael J. Fischer was born on April 20, 1942, in Ann Arbor, MI. He received the B.S. degree in mathematics from the University of Michigan, Ann Arbor, in 1963, and the M.A. degree and the Ph.D. degree in applied mathematics from Harvard University, Cambridge, MA, in 1965 and 1968, respectively.

He is currently Professor of Computer Science at Yale University, New Haven, CT, and has previously taught at Carnegie-Mellon University, the Massachusetts Institute of Technology, and the University of Washington. He is an Area Editor of the *Journal of the Association for Computing Machinery* and is serving on the Editorial Boards of several other journals. His current research interests include theory of distributed systems, reliability, and computational complexity.

Dr. Fischer is a member of the Association for Computing Machinery, the American Mathematical Society, the Society for Industrial and Applied Mathematics, Phi Beta Kappa, and Phi Kappa Phi.

Nancy D. Griffeth, photograph and biography not available at the time of publication.



Nancy A. Lynch received the B.S. degree in mathematics from Brooklyn College, Brooklyn, NY, in 1968, and the Ph.D. degree in mathematics from the Massachusetts Institute of Technology, Cambridge, in 1972.

She is presently Associate Professor of Information and Computer Science at Georgia Institute of Technology, Atlanta, and Visiting Associate Professor of Computer Science at M.I.T. She has also been on the mathematics faculty at Tufts University and the University of Southern California. Her general research interests are distributed computation and theoretical computer science. More specific interests include formal modeling and complexity analysis of distributed algorithms, and design of algorithms for concurrency control and network resource allocation.