Group Communication

Danny Dolev and Dalia Malki

The Transis Approach *to High Availability Cluster Communication*

A unique large-scale multicast service designed for partitionable operation is examined here.

In the local elections system of the municipality of "Wiredville",[1] several computers were used to establish an electronic town hall. The computers were linked by a network. When an issue was put to a vote, voters could manually feed their votes into any of the computers, which replicated the updates to all of the other computers. Whenever the current tally was desired, any computer could be used to supply an up-to-the-moment count.

On the night of an important election, a room with one of the computers became crowded with lobbyists and politicians. Unexpectedly, someone accidentally stepped on the network wire, cutting communication between two parts of the network. The vote counting stopped until the network was repaired, and the entire tally had to be restarted from scratch.

This would not have happened if the vote-counting system had been built with partitions in mind. After the unexpected severance, vote counting could have continued at all the computers, and merged appropriately when the network was repaired.

The "Wiredville" story illustrates some of the finer points that motivated our work in the Transis project [1], a large-scale multicast service designed with the following goals:

[1] The story is based on a real event, but names and details have been changed.

- Tackling network partitions and providing tools for recovery from them. Transis was designed to support partitionable operation, in which multiple network components that are (temporarily) disconnected from each other operate autonomously. When network partitions occur, as in "Wiredville" and in more complicated situations, Transis provides enhanced facilities for an application programmer to construct applications that operate consistently in multiple components of a partitioned network, and to merge these components gracefully upon recovery.
- Meeting the needs of a large network through a hierarchical communication structure, with gateways selectively filtering messages among domains.
- Exploiting the available network multicast within each local-area network (LAN) and providing fast cluster communication. Transis has an efficient protocol for reliable multicast, derived from the Trans protocol [16], that employs the Deering IP-multicast mechanism [6] for disseminating messages using selective hardware-multicast. Coupled with a network-based flow control mechanism we developed, the protocol provides high-throughput group communication.

The Transis approach provides the required flexibility for building diverse fault-tolerant applications, *some of which benefit from continued partitioned operation, and some of which do not.*

Partitionable Operation

The Transis approach distinguishes itself in allowing *partitionable operation* and in supporting consistent *merging* upon recovery. The partitioning of a group results in several disjoint components. Any algorithm that depends on the existence of a single component (a *primary component*) in the system is unable to meet the needs of an important class of distributed applications. Our work assumes that the network might partition and seeks semantics that provide the application with accurate information within each partitioned component.

Our approach is substantially different from similar systems that existed before Transis was launched. For example, the Isis system designates one of the components as *primary*, and shuts down the nonprimary components [3, 4]. During the period prior to shutdown and before the partition is detected, it is possible for a non-primary component in Isis to continue operation, and to perform operations inconsistently with the primary component. Moreover, if the primary component ceases to exist (as, provably, cannot be prevented [5]), then the entire system blocks until it can be reestablished.

Another approach, taken in the Trans/Total system [16], allows the system to continue operation only if enough processors are operational and connected to maintain the resiliency requirement. In the Amoeba system [9], a partitioned group may continue operation within multiple components, unless the user specifies otherwise. However, the system provides no means for merging the components upon recovery.

Partitionable operation is advantageous in increasing the availability of service. In various environments (e.g., wireless networks), communication failures frequently occur and the primary component may be lost. For example, a system of four machines, denoted *A*, *B*, *C*, and *D*, may shift from a {*A*, *B* }, {*C*, *D* }) configuration to a ({*A*, *C* }, {*B*, *D* }) configuration, and back. If this occurs, information exchanged between *A* and *B* in the first configuration can reach the rest of the system while in the second configuration. Thus, partitionable operation may succeed in diffusing the information gradually. On the other hand, a primary component model would deny progress in this scenario.

However, the main concern with the existence of multiple active components is that inconsistent operations may occur in different parts of the system. Some applications can cope with relaxed consistency and allow actions to be completed within disconnected components. For example, in the "Wiredville" story, each component could count votes presented to it separately, since the tally is additive. Other examples are Command, Control, and Communication applications, where it is more important to present the user with up-to-date information rather than trying to maintain systemwide consistency while preventing the user from getting access to any information during partitions. The caveat is that allowing inconsistencies may eventually require the user's application to reconcile information once the network recovers. Another example is an airline reservation system that may allow booking to be performed in detached components and suffer some margin of overbooking. More sophisticated protocols (such as the replication protocol in [12]) may allow gradual diffusion of conflict-prone operations in partitionable environments.

The Transis approach provides the required flexibility for building diverse fault-tolerant applications, some of which benefit from continued partitioned

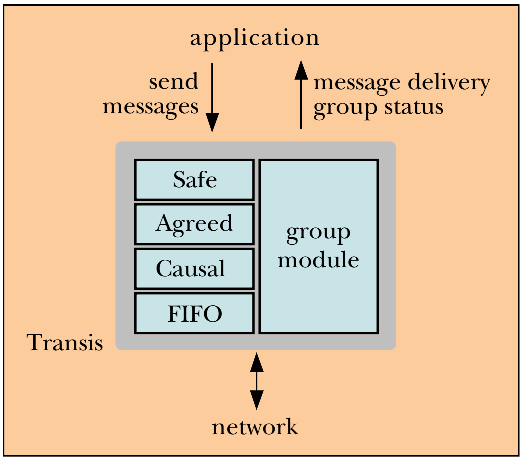


Figure 1. The system model structure

operation, and some of which do not. For the latter kind of application, it is possible to prevent multiple components from coexisting. Today, the Transis approach to partitionable operation has been adopted in other systems, and we have been collaborating for several years with the developers of the Horus system [19] and the Totem system [17], who have incorporated some of our ideas into their architectures.

The Group Service

We begin the description of the Transis system with a general overview of the group communication service. Transis provides *transport* level multicast communication services. As shown in Figure 1, it resides below the user application and above the network layer.

We use the term *service* to refer to the collective work of corresponding modules in all the machines in the network. Thus, *group service* refers to the work of the collection of group modules.

In Transis, the group service is a manager of group messages and group views. Each group module maintains a *local view* (a list) of the currently connected and operational participants in the network. Each local view has a certain lifetime, starting when it is initially reported to the application and ending when its composition changes through a *view change* event (whereby members leave it or join it). In addition to regular views, a group module reports *hidden views* to the application. A hidden view has the same composition as a regular view, but is denoted "hidden." Intuitively, it indicates to the user that the view has failed to form but may have succeeded to form at another part of the system (the next section elaborates on the utility of hidden views in some applications).

The local view provided by a group service may not always be accurate, since failure detection in a realistic system is generally unreliable. At any point in time, local views at different machines may be identical, overlapping, or nonintersecting. However, the significance of local views as well as hidden views is in the interrelation between views at different machines. The behavior of the collective group service is defined through the requisites given informally in Figure 2 (for a full specification and a description of a group membership protocol that specifies it, see [7]).

Within the lifetime of each view, the group module delivers multicast messages. Transis supports several types of multicast services: **FIFO** multicast guarantees sender-based FIFO delivery order. **Causal** multicast preserves the potential causal order among messages (as defined in [13]). **Agreed** multicast enforces a unique order among every pair of messages in all of their destinations.[2] A **safe** multicast guarantees a unique order of message delivery, and in addition, delays message delivery until the message is acknowledged by the transport layers in all of the machines in the group, thus guaranteeing delivery atomicity in case of communication failures.

View change reporting and message delivery at different members of the group are coordinated by the group service. This principle is called *virtual synchrony* [4] and is extended in Transis to partitionable environments. Intuitively, the virtual synchrony principle guarantees that a local view reported to any member is reported to all other members, unless they crash. In case of partitions, we guarantee virtually synchronous behavior within each isolated component and coherent behavior when components merge. The net effect is that the application builder is presented with coherent system behavior, and can employ less complex failure-handling code within the application.

The Partitionable Operation Methodology

After a network partitions, here is what we would like the situation to be in any distributed application:

- At least one component of the network should be able to continue making updates. The situation should be the same whether other components are down or the network is detached.
- Each machine should know about the update messages that reached all of the other machines before they were disconnected.
- Upon recovery, only the missing messages should be exchanged to bring the machines back into a consistent state.

Unfortunately, not everything in our wish-list is possible. When there is no bound on the duration of message passing in the network (the system is *asynchronous*), Chandra et al. have shown that it is impossible to maintain a primary component, whose membership is known and agreed upon in all cir-

[2] Some prior work refers to this as *total* or *atomic* ordering.

cumstances [5]. The impossibility of maintaining a primary component means that progress might completely halt in applications that disallow all but one part of the system to perform operations (e.g., when the primary component makes decisions or answers on behalf of the system). Thus, in some applications, the first item in the preceding list might be impossible to accomplish.

Furthermore, knowing which messages have been delivered by all of the computers at any point in time requires *instantaneous knowledge* about message delivery and is provably impossible to achieve in an asynchronous distributed environment [8]. Thus, the second item is impossible.

Therefore, it is inevitable that we must operate with some margin of uncertainty. This, however, does not imply that the situation need be completely chaotic.

First, not all applications require that progress is made exclusively in a primary component. Recall the "Wiredville" town hall example from the introduction: It is possible to allow all parts of the partitioned network to continue counting separately, and upon recovery, to merge them (avoiding double-counting of votes). Simple unordered diffusion of messages between previously detached components can be accomplished by *gossiping* after partitions are mended.

Second, when local views merge in Transis, the diffusion of messages can be done very efficiently. After merging several previously-detached components, each component can be represented by a single member. So for example, in the "Wiredville" town hall application, a single computer in each half of the network could replay messages upon merging on behalf of its entire component. The set of messages delivered within the component before merging, within the duration it was detached from the rest of system, can be replayed by this representative member alone. If further failures occur during merging, then representatives must be chosen out of any previously connected component for which message-replaying has not been completed. Due to virtual synchrony, it is guaranteed that all of the other members in the component have delivered during the detached period the same set of messages as the representative.

In the "Wiredville" application, partitioning is exceptionally easy to handle since the tally is additive. Unfortunately, many applications can allow updates only within a primary component. For this kind of application, our approach provides support for recovering a primary component if it has been lost. In order to maintain progress in face of partitions, members in all of the components can remain operational and wait to merge with a primary component or to generate a new one if it is lost. Members of different components have information allowing them to totally order past view change events. Thus, it is possible to track a primary component and eventually recover from the possible loss of it (even in the case of a total network failure). The difficulty in recovering from the loss of a primary component is that the recovered machines may be in a symmetrical situation after the recovery, as illustrated in the following example. Transis assists the application developer in such situations by reporting hidden-views, and thus breaking the apparent symmetry.

Figure 2. A framework for partitionable group service

Terminology:

Local View: A list of machines, reported to a member. The local view is modified by a *view change* event.

Messages: Messages are *multicast* to the group, with varying requirements: **FIFO, causal, agreed, safe.**

Requisites:

Self Inclusion: A local view at a machine always includes the machine itself.

Same Order: View changes occur in the same order at all their overlapping members.

Virtual Synchrony: Between consecutive view change events, the same set of messages is delivered by all overlapping members.

In addition, message delivery maintains the multicast requisites, **FIFO, causal, agreed** or **safe.**

Uniformity: A view change reported to a member is reported to all of the other members, as either a regular or a *hidden* view change (a *hidden view* has the same view composition as a regular one, but is marked "hidden").

A hidden view change reported to a member may be reported as a regular view change elsewhere, in an execution that is otherwise identical to this member's.

Liveness: A machine that does not respond to messages sent to it is removed from the local view of the sender within a finite amount of time. Every connected set of machines eventually forms a common view, reported to all of the members.

Safety: Machines are removed from a view only by the *Liveness* property.

We illustrate the breaking of symmetry through an example scenario (see Figure 3).

There are three machines in the system, denoted *A*, *B*, and *C*, and each majority of two or more machines is considered primary. *A*, *B*, and *C* start as one connected component { *A*, *B*, *C* }. Later, *C* splits from { *A*, *B* }. A reports the local view { *A*, *B* }, but *B* detaches from *A* before *B* succeeds in reporting it. Following that, *B* connects back with *C*, and *C* reports the view { *B*, *C* }, *B* crashes, and *C* joins with *A*. First, think of this scenario without the hidden (dashed) view { *A*, *B* } reported at *B*. *A* and *C* seem to be in symmetrical state; *A*'s history of view change events consists of { *A*, *B*, *C* }, { *A*, *B* } and *C*'s history is { *A*, *B*, *C* }, { *B*, *C* }. Unless *B* had passed information to *C* about the possible hidden view { *A*, *B* }, *A* and *C* cannot determine which has a more up-to-date state, since *B* is missing. On the other hand, if *B* is indeed informed of the hidden view { *A*, *B* }, then it can carry this information into the next connected component { *B*, *C* }.

Exactly how the hidden view information is used is application-dependent. The interested reader is referred to [2, 11, 12] for applications that use the Transis partitionable group service.

The Hierarchical Broadcast Domain

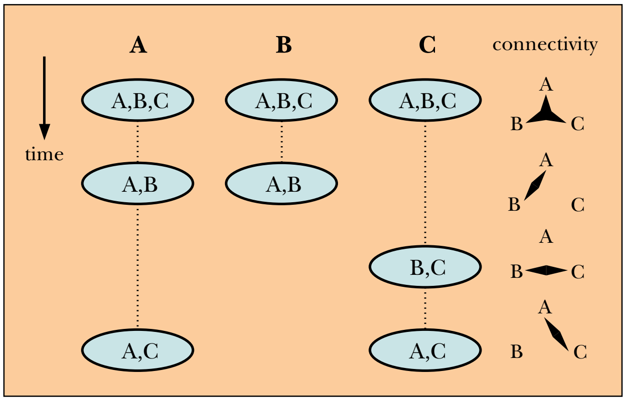
We have previously discussed the Transis partitioning mechanism, which is crucial in large and dynamic settings. A large network also presents challenges due to the diversity of communication media and its structure. Transis provides high-throughput communication through protocols that exploit the underlying network structure and was a pioneer in demonstrating high-performance communication in practice. We model a wide-area network (WAN) as a hierarchy of *multicast clusters*, as depicted in Figure 4. Each multicast cluster represents a domain of machines capable of communicating via broadcast hardware or via selective-multicast hardware (e.g., the Deering IP-multicast [6]). In reality, such a cluster could be within a single local-area network (LAN) or multiple LANs interconnected by transparent gateways or bridges.

Clusters are arranged in a hierarchical group structure, with representatives from each local domain participating in the next level up the hierarchy. Each level of the hierarchy is a group domain that maintains the group services (discussed in the section "The Group Service") internally. Each cluster employs the Trans-based reliable message recovery engine, described in the next section, which achieves very high message throughput. Between clusters, other forms of communication are used. In each cluster, a representative *selectively* filters messages leaving the cluster and forwards incoming messages into the cluster.

There are several key ideas in the design of the hierarchical structure:

• Each level of the hierarchy abstracts the levels below it and maintains group services at the level itself. In this way, group services are made scalable and maintainable in a WAN.

Figure 3. Breaking the symmetry between A and C



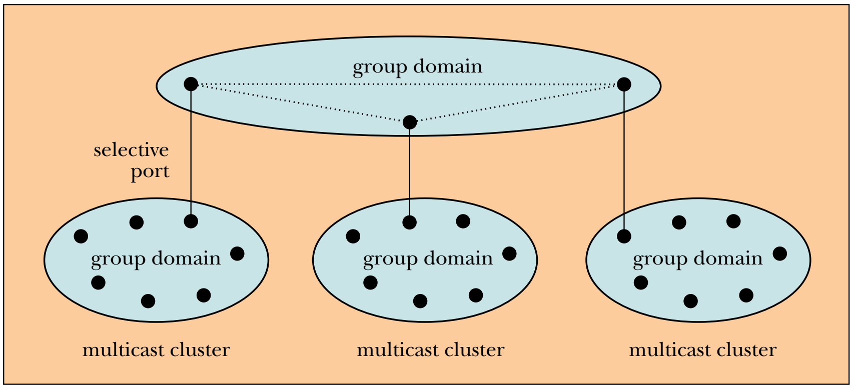


Figure 4. The Transis communication model

A large network also presents challenges **due to the diversity of communication media and its structure.**

- Each port selectively passes messages to and from the clusters it belongs to, thus avoiding flooding a WAN with local traffic.
- The multicast protocol within a multicast cluster exploits the available broadcast hardware to achieve high-throughput group communication.

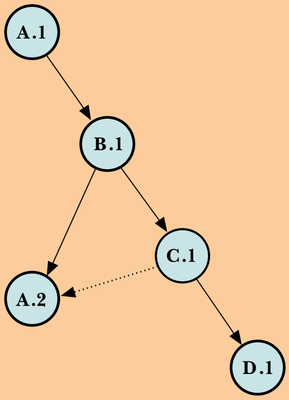
A High-Performance Reliable Multicast Engine

The transport service within a multicast cluster employs the reliable multicast engine presented (briefly) in the sidebar "The Reliable Multicast Communication Protocol" (a detailed description can be found in [14]). There are several important principles that underlie our design:

The Reliable Multicast Communication Protocol

The reliable transport engine within a local area network, derived from the Trans protocol [16], utilizes the available broadcast hardware for disseminating messages in a single transmission. Acknowledgments are piggybacked on regular messages, and are thus also broadcast once. Messages form a "chain" of acknowledgments, which implicitly acknowledge former messages in the chain. We denote by *X.n* the *n*'th message from machine *X*, and a piggybacked acknowledgment for it by $\overset{X.n}{\hookrightarrow}$. For example, we could have the following scenario on the network:

$$A.1\,;\ \overset{A.1}{\hookrightarrow} B.1\,;\ \overset{B.1}{\hookrightarrow} C.1\,;\ \overset{C.1}{\hookrightarrow} D.1\,;\ \ldots$$



Machines in a multicast cluster can recognize message losses by analyzing the received message chains. For example, machine A could recognize that it lost *C*.1 after receiving the sequence: *A*.1; $\overset{A.1}{\hookrightarrow}$ *B*.1; $\overset{C.1}{\hookrightarrow}$ *D*.1. Therefore, A emits a *negative-ACK* (*A*.2, dashed arrow) on message *C*.1, requesting its retransmission. Machine A **defers** its acknowledgment of *D*.1 until *C*.1 is recovered, since messages that follow "causal holes" are not incorporated for delivery until the lost messages are recovered. Meanwhile, A can acknowledge *B*.1. In this way, the acknowledgments form the causal relation among messages directly.

- In systems that do not lose messages very often, it is preferable to use a negative acknowledgment-based protocol. Thus:

  – Messages are not retransmitted unless explicitly requested to do so by means of a negative acknowledgment.

  – Positive acknowledgments, required for determining the arrival of messages, are piggybacked onto regular messages. If no regular message is transmitted, then periodically, an empty message containing only acknowledgments and an "I am alive" indication will be sent.

Modern networks exhibit extremely low message-loss rates, hence these mechanisms have become widely used (see, for example, [3, 10, 16, 18]).

- Detection of lost messages must occur as soon as possible. Suppose that machines A, B, and C send successive messages. If machine D uses sender-based FIFO order to detect message loss, and misses the message from A, then it will not detect the loss until A transmits another message. On the other hand, if there are additional relationships between messages sent by different machines, then as soon as B transmits its message with a reference to A's message, D can possibly detect the loss of A's message. Early detection saves buffer space by allowing prompt garbage collection, regulates the flow, and prevents cascading losses.
- Under high communication loads, the network and the underlying protocols can be driven to high loss rates. For example, based on experiments using UDP/IP communication between Sun Sparcstations interconnected by 10Mb Ethernet, we found that under normal load, the loss rate was approximately 0.1%, but under extreme conditions, the loss rate went up to 30%. Such high loss rates would make the recovery of lost messages costly, and can cause an avalanche effect further increasing the load on the communication medium. To prevent this, it is crucial to control the flow of messages in the network. In the Transis system, we employ a flow control method called the *network sliding window* (see [1, 15] for further details).

Details on the performance of the reliable multicast engine are provided in [14].

Conclusion

Transis is a transport-level group communication service that simplifies the development of fault-tolerant distributed applications, in that it presents a coherent behavior to the user upon failures. In a world of growing dependency on computers, the ability to continue operation in a dynamic environment is crucial. We were able to extend a successful approach for developing fault-tolerant distributed applications using group communication—limited by primary-component assumptions—into large-scale environments where challenges like partitioning and exploiting the network hardware abound. Transis supports partitionable operation and provides the strictest semantics possible in the case of network failures. Through a precise definition of partitionable operation, we provide the means of recognizing the difficulties inherent in distributed coordination and of reconciling wishes with impossibilities, such as the ones cited previously in the section "The Partitionable Operation Methodology."

The process of converting uniprocessor software to

In a world of growing dependency on computers, *the ability to continue operation in a dynamic environment is crucial.*

a distributed fault-tolerant program is not made automatic by our tools. Future development in this area must better explore programming methodologies integrated with group communication frameworks. In Transis, we have emphasized methodologies and enhanced tools for taking advantage of the partitionable membership service. We are now taking the next step by pursuing the development of services that implement higher-level building blocks (e.g., replication services that automatically reconcile merged partitions, presented in [2] and [12]).

Acknowledgments

We acknowledge the contribution of members of the Transis project: Yair Amir and Shlomo Kramer of the original Transis team who designed and shaped Transis, and those who currently extend the system: David Breitgand, Gregory Chockler, Yair Gofen, Nabil Huleihel, Idit Keidar, and Roman Vitenberg. Various projects of many other students have helped in bringing Transis to its current state.

Various components of the Transis project were funded in part by the Ministry of Science of Israel, grant number 0327452, by the German Israel Foundation (GIF), grant number I-207-199.6/91, and by the United States-Israel Binational Science Foundation, grant number 92-00189.

The Transis home page is http://www.cs.huji.ac.il/papers/transis/transis.html

References

1. Amir, Y., Dolev, D., Kramer, S., and Malki, D. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing* (July 1992), 76–84.
2. Amir, Y., Dolev, D., Melliar-Smith, P.M., and Moser, L.E. Robust and efficient replication using group communication. Tech. Rep. CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.
3. Birman, K.P. The process group approach to reliable distributed computing. *Commun. ACM 36*, 12 (Dec. 1993), 36–53.
4. Birman, K.P. and Joseph, T. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Annual Symposium on Operating Systems Principles* (Nov. 1987), 123–138.
5. Chandra, T.D., Hadzilacos, V., and Toueg, S., and Charron-Bost, B. On the impossibility of group membership. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing.* (1996). To be published.
6. Deering, S.E. Host extensions for IP multicasting. RFC 1112, SRI Network Information Center, Aug. 1989.
7. Dolev, D., Malki, D., and Strong, H.R. An asynchronous membership protocol that tolerates partitions. Tech. Rep. CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.
8. Halpern, J.Y. and Moses, Y. Knowledge and common knowledge in a distributed environment. In *Proceedings of the 3d Annual ACM Symposium on Principles of Distributed Computing* (1984), pp. 50–61.
9. Kaashoek, M.F. and Tanenbaum, A.S. Group communication in the amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems* (May 1991), pp. 882–891.
10. Kaashoek, M.F., Tanenbaum, A.S., Hummel, S.F., and Bal, H.E. An efficient reliable broadcast protocol. *Operating Systems Review 23*, 4 (Oct. 1989), 5–19.
11. Keidar, I. and Dolev, D. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the Symposium on Principles of Database Systems* (May 1995), pp. 245–254.
12. Keidar, I. and Dolev, D. Efficient message ordering in dynamic networks. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing.* (1996). To be published.
13. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558–565.
14. Malki, D. Multicast communication for high availability. Ph.D. dissertation, Inst. of Computer Science, The Hebrew University of Jerusalem, 1994.
15. Malki, D., Amir, Y., Dolev, D., and Kramer, S. The transis approach to high availability cluster communication. Tech. Rep. CS94-14, Inst. of Comp. Sci., The Hebrew University of Jerusalem, June 1994.
16. Melliar-Smith, P.M., Moser, L.E., and Agrawala, V. Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distributed Syst. 1*, 1 (Jan. 1990) 17–25.
17. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., and Lingley-Papadopoulos, C.A. Totem: A fault-tolerant multicast group communication system. *Commun. ACM 39,* 4 (Apr. 1996).
18. Peterson, L.L., Buchholz, N.C., and Schlichting, R.D. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst. 7*, 3 (Aug. 1989), 217–246.
19. Van Renesse, R., Birman, K.P., and Maffeis, S. Horus: A flexible group communication system. *Commun. ACM 39,* 4 (Apr. 1996).

About the Authors:

DANNY DOLEV is a professor at the Institute of Computer Science, the Hebrew University of Jerusalem. **Author's Present Address:** Institute of Computer Science, the Hebrew University of Jerusalem, Jerusalem 91904, Israel. email: dolev@cs.huji.ac.il

DALIA MALKI is a member of the technical staff at AT&T Bell Laboratories. **Author's Present Address:** AT&T Bell Laboratories, 600 Mountain Ave., Murray Hill, NJ 07974. email: dalia@research.att.com