GROUP COMMUNICATION

In distributed computer systems, several or many computers cooperate to perform an application, and information may be replicated on several computers. Replication of information may be required to provide fault tolerance or increased availability after a fault. Replication may also be used to reduce the time required to access the information by providing local or nearby copies of it.

Much of the difficulty of programming distributed applications derives from the need to maintain the consistency of replicated information, in the presence of asynchrony and faults. When the replicated information has become inconsistent, the application programming and/or human intervention required to restore consistency can be quite difficult. In general, it is easier to maintain consistency than to restore it.

Inconsistency can arise, for example, if a process holding one replica of the data fails to receive a message and thus does not apply an update to that replica. This is an example of an omission inconsistency, as shown in Fig. 1(a).

Inconsistency can also arise, as shown in Fig. 1(b), if one process creates a new database entry and transmits an instruction in a message broadcast to the other processes that they should also create that entry. A second process receives the message and generates an update for the new entry, communicating the update in a message broadcast to the other processes. If one of the processes holding a replica of the database receives the second message before the first, it may be unable to handle the message. This is an example of a causal ordering inconsistency.

Inconsistency can also arise when two or more processes try to claim a resource, as shown in Fig. 1(c). The requests for the resource are sent in broadcast messages to the multiple processes holding the resources, with the first claimant getting the resource. Multiple processes are needed to manage the resource to ensure continued operation, if a process should fail. If those processes receive the messages in different orders, they may grant the resource to different requesters. This is an example of a total ordering inconsistency.

Group communication systems provide message ordering and delivery services that assist the application programmer in avoiding these inconsistencies.

- Reliable delivery of messages ensures that all messages broadcast to a group of processes are delivered to all members of the group, thus precluding omission inconsis-

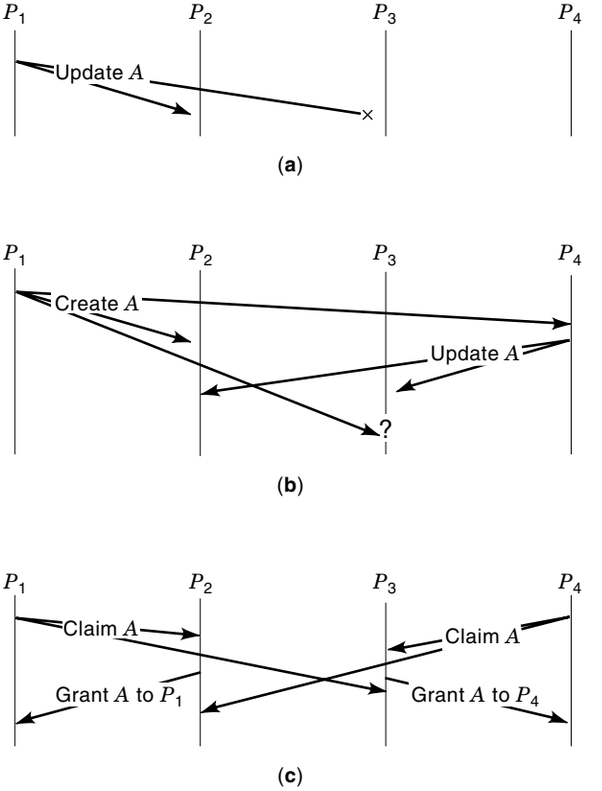

Figure 1. Examples of (a) omission inconsistency, (b) causal ordering inconsistency, and (c) total ordering inconsistency.

tencies. This also requires that messages are delivered exactly once.

- Causally ordered delivery of messages ensures that, if a message depends on a prior message, then no process receives that message before it receives the prior message. This precludes causal ordering inconsistency.
- Totally ordered delivery of messages ensures that any pair of messages delivered by two or more processes are delivered in the same order by those processes, thus precluding total ordering inconsistency.

These message delivery services are discussed further in the next section, titled "Formal Models and Definitions," and algorithms for achieving these services are given in the section titled "Message Delivery Algorithms."

Group communication systems also provide membership services that maintain the membership of the group, add new processes to the group, remove departing or faulty processes from the group, and report changes in the group membership to the application. If the network partitions into multiple components with no communication among them, membership algorithms are confronted with conflicting objectives. If the application must maintain a single consistent state of its data, then the membership algorithm must form a single primary membership component. If, however, all processes must continue operation even when disconnected, then the membership algorithm must form multiple disconnected memberships. Group membership algorithms for primary component membership and for partitionable membership are described in the section titled "Membership Algorithms."

Group membership services depend on the detection of faulty processes. Unfortunately, in an asynchronous model of computation, it is impossible to distinguish between a process that has crashed and one that is merely slow. Consequently, most group communication systems depend on unreliable fault detectors (1) that detect faulty processes but that are unreliable because they might also suspect a slow process to be faulty. Unreliable fault detectors are discussed in the section titled "Fault Detectors."

To enable a new member of the group to participate meaningfully in the group, it is necessary for a process that is already a member of the group to assemble and transfer state information to the new member. If a message has been processed before the state information is transferred, and if that same message is processed by the new process after it receives the state, an incorrect state can result. An incorrect state can also result if a message has not been processed before the state is assembled and transferred, but the new process determines that the message is an old message and thus ignores it. Consequently, processes must agree on which messages are delivered and processed before the membership change so that their effects are included in the transferred state, and which messages are delivered after the membership change so that they are processed by the new process. Virtual synchrony (2), discussed in the section titled "Formal Models and Definitions," ensures that the processes agree on which messages precede a membership change and which follow it.

Group communication systems can achieve high performance by exploiting available broadcast mechanisms at the physical and network layers in local- and wide-area networks. Implementing broadcasts by multiple point-to-point messages results in poor performance. To achieve high performance, group communication systems must address the issues of buffer management and flow control. When several, or many, processes broadcast messages almost simultaneously, the communication medium and the buffers can quickly become exhausted, resulting in lost messages. Broadcasts to many destinations can also result in large numbers of acknowledgments returning to the source, again exhausting the available resources. The best group communication protocols carefully manage the flow of messages to reduce network contention and ensure that buffers are available at the destinations. Such systems deliver messages reliably to many destinations with performance as good as that of point-to-point protocols operating between a single source and a single destination.

Fully integrated group communication protocols (3) address these issues in a unified fashion. Such systems are highly efficient and provide a comprehensive set of services at little or no extra cost. A standard set of services reduces the amount of skill required to exploit the protocol effectively. On the other hand, group communication toolkits (4) provide microprotocols that the user assembles into a protocol that is customized for the application. A custom protocol avoids the costs of services not needed by the application but, instead, incurs the costs of interfaces and mechanisms that are designed to work independently. Furthermore, the user may find it difficult to choose an appropriate set of microprotocols for the particular application.

FORMAL MODELS AND DEFINITIONS

The models, definitions, and algorithms described here can be used for sets of processors or for sets of processes executing on multiple processors. In this article, we refer to processes but processors can be substituted throughout.

A *distributed system* is a collection of processes that communicate by messages. In a *group communication system,* the processes are organized into *groups* and messages are *broadcast* to all members of the group. In contrast, messages that are *multicast* may be sent to some of the processes but not necessarily all of them. We distinguish between the terms receive and deliver. A process *receives* messages that were broadcast by the processes, and a process *delivers* messages to the application.

Distributed systems can be classified as synchronous or asynchronous. In an *asynchronous* system, it is not possible to place a bound on the time required for a computation or for the communication of a message. Even though processes may have access to local clocks, those clocks are used only for local activities; they are not synchronized and are not used for global coordination. The advantage of the asynchronous model is that the algorithms can be designed to operate correctly without regard to the number of individual processes or the timing characteristics of the processes or of the communication medium. The disadvantage is that performance characteristics, such as message delivery latency, are necessarily probabilistic, and performance analysis and prediction are difficult.

In contrast, the synchronous model requires that processes have access to local clocks that are synchronized across the system with a known bound on the skew between clocks, and

that computation and communication operations complete within specified periods of time. The synchronous model is particularly suited to hard real-time systems and has the advantage that algorithms are deterministic and less complex. The disadvantage is that conservative assumptions are required to approximate synchronous operation in the real world and the resulting system may be inefficient.

The timed asynchronous model (5) closely resembles the asynchronous model but includes, in addition, a requirement that eventually there will exist an *interval of stability* within which computation and communication operations complete successfully within a specified time bound. For this model, the algorithms are generally similar to those for the asynchronous model but are simpler because they need to guarantee termination only within the stability interval. The delay until an interval of stability is reached will, however, probably be longer than the delay until termination of the asynchronous algorithm. The timed asynchronous model trades simpler algorithms against longer termination times.

Fault Models

In any distributed system model, a process that is *nonfaulty* (*correc*t) performs the steps of the algorithms according to the specifications. The behavior of a faulty process depends on the particular fault model adopted.

In the *fail-stop* and *crash* models, a faulty process takes no actions (i.e., it sends no messages and ignores all messages that it receives). These two models differ in that a fail-stop process reports that it has failed, whereas a crash process does not. Both models typically assume that a faulty process never recovers or, if it recovers, is regarded as a new process.

Other models allow a faulty process to be repaired and to be reconfigured into the system. When a process recovers, the process knows its process identifier and can retrieve the data it had written to persistent storage, such as disk, before it failed.

In the more general *Byzantine* fault model, processes can exhibit arbitrary and even malicious behavior, such as generating incorrect messages or sending different messages to different processes that purport to be the same message. In general, it is more difficult to develop algorithms that are resilient to Byzantine faults than to fail-stop or crash faults.

Most fault models admit communication faults in the form of *message loss,* which is caused by corruption by the medium or buffer overflow in the intermediate switches or at the destinations. Some fault models also admit *network partitioning* faults, which split the system into two or more components so that processes in the disconnected components cannot communicate with one another.

The synchronous model augments this fault model in that any computation or communication operation that does not complete within its specified time bound constitutes a fault. Similarly, any excessive skew between clocks constitutes a fault.

Impossibility Results

The problem of maintaining consistent message delivery and membership in a system subject to faults is related to the problem of achieving consensus in such a system. Fischer, Lynch and Paterson (6) have shown that it is impossible to devise a deterministic algorithm that can guarantee to achieve consensus in finite time in an asynchronous distributed system, even if communication is reliable and only one process can crash. Chandra and Toueg (1) have shown, however, that consensus is possible in an asynchronous distributed system that is subject to crash faults if an unreliable fault detector is provided. Randomized algorithms can also be used to achieve consensus in an asynchronous distributed system that is subject to faults (7,8).

In practical systems with unreliable communication, even the reliable delivery of a single message cannot be guaranteed. There is also a nonzero probability that all of the processes will fail. Impossibility results should not, however, be regarded as a proof that asynchronous distributed systems cannot be built. Rather, they are a reminder that the algorithms must be robust against unfortunate sequences of asynchronous events and faults, so that message ordering and membership decisions can be reached in a reasonable time with a high probability.

Message Delivery Services

We now consider various message delivery services. The most basic type of message delivery service is *unreliable* message delivery, which provides a best-effort service with no guarantees of message delivery or of the order of message delivery. Unreliable message delivery is used for applications such as audio and video streaming. Many other applications, however, require one of the more stringent message delivery services, which are described here and shown in Fig. 2.

Reliable Delivery. *Reliable* message delivery requires that if any nonfaulty (correct) process receives a message, then all nonfaulty processes eventually receive that message, possibly after multiple retransmissions. Reliable delivery can be achieved only probabilistically in the presence of an unreliable communication medium that repeatedly loses messages.

Source Ordered Delivery. *Source ordered* delivery, or First-In First-Out (*FIFO*) delivery, requires that messages from a particular source are delivered in the order in which they were sent. Source ordered delivery is appropriate for multimedia streaming and data distribution applications.

Causally Ordered Delivery. *Causally ordered* delivery (9) satisfies the following two properties:

- If process P sends message M_1 before it sends message M_2 and process Q delivers both messages, then Q delivers M_1 before it delivers M_2.
- If process P receives message M_1 before it sends message M_2 and process Q delivers both messages, then Q delivers M_1 before it delivers M_2.

Taking the transitive closure of this "delivers before" relation yields a partial order on messages. A partial order, denoted by $\sim$, satisfies the following properties:

- *Antireflexive:* $M \nsim M$.
- *Antisymmetric:* If $M_1 \sim M_2$, then $M_2 \nsim M_1$.
- *Transitive:* If $M_1 \sim M_2$ and $M_2 \sim M_3$, then $M_1 \sim M_3$.

Delivery of messages in causal order precludes causal ordering inconsistency and prevents anomalies in the processing of

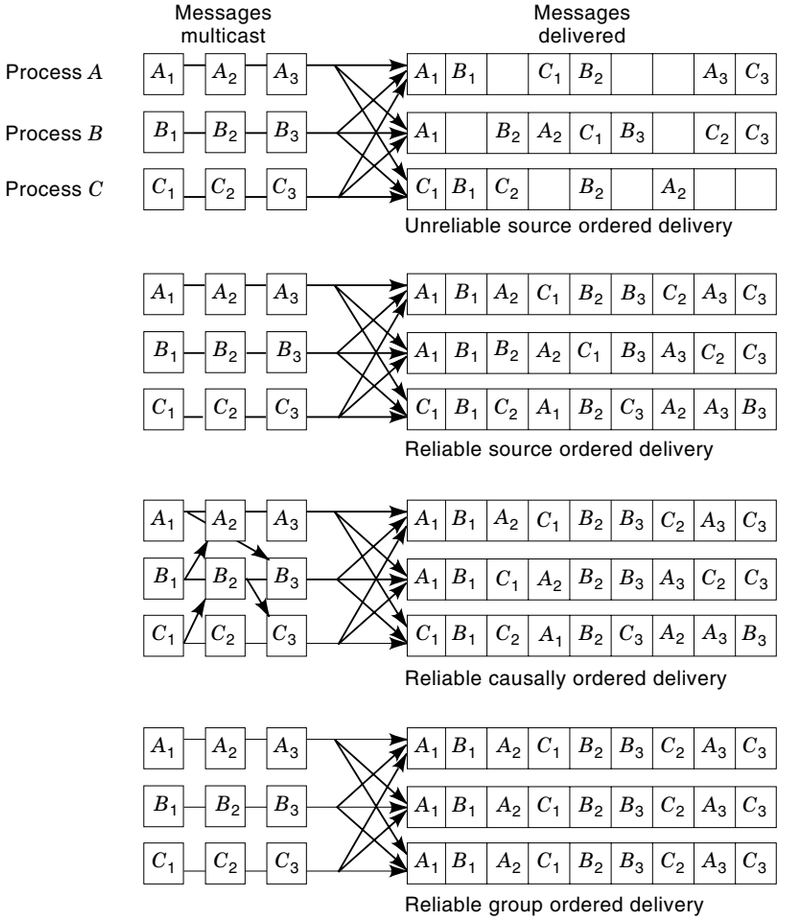

Figure 2. Examples of four types of message delivery services.

data contained in the messages, but it does not alone suffice to maintain the consistency of replicated data.

Group Ordered Delivery. *Group ordered* delivery requires that, if processes P and Q are members of a group G, then P and Q deliver the messages originated by the processes in G in the same total order. A reliable group ordered message delivery service helps to maintain the consistency of replicated data, but inconsistencies can still arise in interactions beween groups.

Totally Ordered Delivery. *Totally ordered* delivery, also called *atomic* delivery, subsumes partially ordered delivery but requires in addition the property:

- *Comparable:* $M_1 \sim M_2$ or $M_2 \sim M_1$.

Thus, totally ordered delivery requires that, if process P delivers message M_1 before it delivers message M_2 and another process Q delivers both messages, then Q delivers M_1 before it delivers M_2. If P and Q are both members of the same process group and M_1 and M_2 are both sent to that group, then group ordered delivery would have sufficed to ensure this property. Typical applications contain hundreds of process groups, and many messages are sent to multiple groups. Totally ordered delivery precludes total ordering inconsistency and is important where systemwide consistency across many groups is required. Thus, more generally, totally ordered delivery implies that there are no cycles in the total order, because the total order is a partial order.

Message Stability. A message is *stable* at a process when that process has determined that all of the other processes in its current membership have received the message. This determination is typically based on acknowledgments of messages. When a process determines that a message has become stable, it can reclaim the buffer space used by the message because it will never need to retransmit that message again. The concept of stability of messages is quite distinct from stability of data, which requires that the data have been written to nonvolatile storage, such as a disk. Delivery of messages only when the messages have become stable is useful, for example, in transaction-processing systems where a transaction must be committed by all of the processes or none of them.

Membership Services

Maintaining the membership of the groups is an important part of group communication systems because algorithms may block if processes are faulty or cannot communicate with one another. An unreliable fault detector can be used to detect apparently faulty processes, to trigger the membership algorithm, and to ensure that the algorithm satisfies liveness requirements (i.e., that decisions will be made and that the sys-

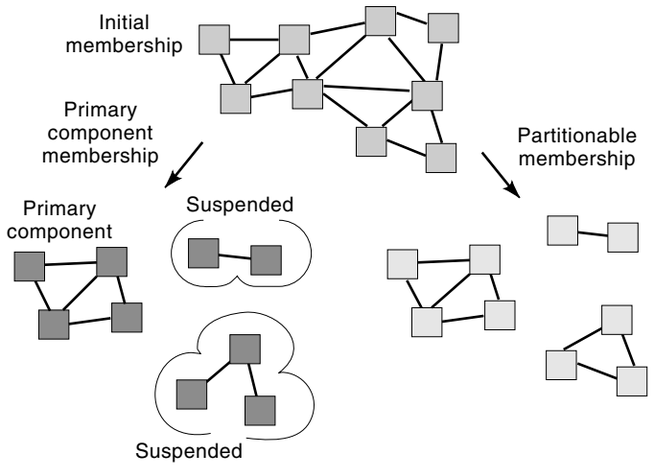

Figure 3. The primary component membership model allows only a single component of a partitioned system to continue to operate, whereas the partitionable membership model allows continued operation in all components.

tem will continue to make progress). The membership algorithm removes apparently faulty processes from the membership and adds new or recovered processes into the membership.

Different group communication systems have adopted different formulations of the membership problem. In the *primary component* model (2), for each process group a single sequence of memberships must be maintained across the distributed system over time, as shown at the left of Fig. 3. In this model, the membership algorithm, upon successive invocations, yields a sequence of memberships over time.

In contrast, the *partitionable membership* model (10) allows multiple disjoint memberships to exist concurrently. At opposite ends of the spectrum are two approaches to the partitionable membership problem, the maximal memberships approach and the disjoint memberships approach. In the *maximal memberships* approach, the memberships are precisely the maximal cliques, and a nonfaulty process may belong to several (perhaps many) concurrent memberships. In the *disjoint memberships* approach, concurrent memberships do not intersect (i.e., each nonfaulty process is a member of exactly one membership at a time, and any pair of processes in a membership can communicate). Thus, each membership is a clique, although not necessarily a maximal clique. The partitionable membership model with disjoint memberships is shown at the right of Fig. 3. Neither the disjoint membership approach nor the maximal cliques approach is ideal, but it is not obvious how an intermediate approach would define the collection of memberships.

An algorithm that solves these membership problems must ensure that the processes in a membership reach agreement on the membership in a finite amount of time. The algorithm should also ensure that faulty processes are eventually removed from the membership and that nonfaulty processes are not removed capriciously so that a trivial membership is not installed when a larger membership could have been installed. Thus, the membership algorithms must ensure the following properties:

- *Agreement:* All processes in the membership agree on the membership set.
- *Termination:* A new membership must be formed within a finite amount of time.
- *Nontriviality:* The agreed upon membership should be appropriate, and nondegenerate if possible.

The appropriateness of the membership may need to be determined by heuristics. For the partitionable membership problem, several existing specifications admit algorithms that yield degenerate memberships by partitioning the membership into singletons even when larger memberships would have been possible. Specification of the partitionable membership problem is an open research topic.

Virtual Synchrony and Extended Virtual Synchrony

Virtual synchrony (2) ensures that view (configuration) changes occur at the same point in the message delivery history for all operational processes, as shown in Fig. 4. Processes that are members of two successive views must deliver exactly the same set of messages in the first view. A failed process that recovers can be readmitted to the system only as a new process. Thus, failed processes are not constrained as to the messages they deliver or their order, and messages delivered by a failed process have no effect on the system. If the system partitions, only processes in one component, the primary component, continue to operate; all of the other processes are deemed to have failed.

Extended virtual synchrony (11) extends the concept of virtual synchrony to systems in which all components of a partitioned system continue to operate and can subsequently remerge and to systems in which failed processes can be repaired and can rejoin the system with stable storage intact. Two processes may deliver different sets of messages, when one of them has failed or when they are members of different components, but they must not deliver messages inconsistently. In particular, if process P delivers message M_1 before P delivers message M_2, then process Q must not deliver message M_2 before Q delivers message M_1, even if the system has partitioned and P and Q can no longer communicate.

Extended virtual synchrony eliminates gratuitous inconsistencies between processes that become disconnected by a partitioning fault. Interestingly, extended virtual synchrony

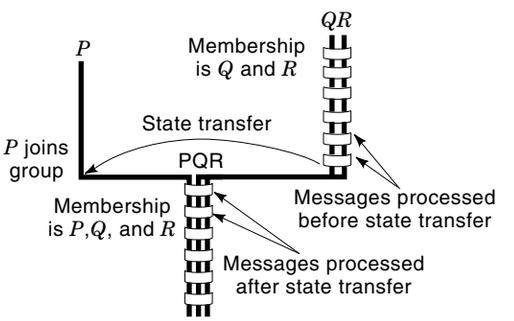

Figure 4. When a membership change brings a new process into the group, the current state of the existing members must be transferred to the new process. Virtual synchrony ensures that all processes agree on which messages precede the transfer of state to a new process and which follow that transfer.

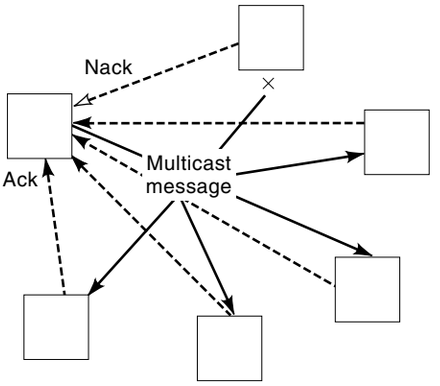

Figure 5. If a process transmits a message to many destinations, it may suffer from an implosion of acknowledgments.

can be guaranteed only if messages are *born ordered,* meaning that the relative order of any two messages is determined directly from the messages, as broadcast by their sources.

MESSAGE DELIVERY ALGORITHMS

We now consider algorithms that provide different types of message delivery, as defined in the section titled "Formal Models and Definitions."

Reliable Delivery Algorithms

Reliable delivery algorithms typically depend on underlying physical or network layer broadcast or multicast mechanisms that provide only an unreliable best-effort service in which messages may be lost. Algorithms that provide a reliable delivery service aim to ensure that every message is delivered to all of the intended destinations. Error detection and retransmission are typically more often used to provide reliable delivery.

Traditional broadcast and multicast algorithms exploit a positive acknowledgment strategy to provide reliable delivery. On receipt of a message, a destination transmits a positive acknowledgment to the source. The source retransmits the message repeatedly until it has received a positive acknowledgment from every destination. Positive acknowledgment algorithms are effective in improving reliability, but they suffer from two problems. First, large numbers of acknowledgments must be transmitted, even when the underlying mechanisms are quite reliable and few messages need to be retransmitted. Second, if there are many destinations, the source must receive and process many acknowledgments for each message that it transmits, resulting in substantial processing overhead, as shown in Fig. 5. This is called the *ack implosion* problem.

Consequently, most reliable broadcast and multicast algorithms use negative acknowledgments to achieve reliable delivery. The source transmits messages with sequence numbers. Destinations detect missing messages by gaps in the sequence numbers and transmit, to the source, negative acknowledgments that list the missing messages. On receipt of a negative acknowledgment, the source retransmits the requested messages. Negative acknowledgment algorithms can be used to achieve reliable delivery, even though they use fewer acknowledgment messages.

Reliable delivery algorithms based on negative acknowledgments suffer from two problems. First, if a message is not delivered to several destinations (e.g., because it was lost as a result of buffer overflow at an intermediate switch), all of those destinations will transmit a negative acknowledgment when one would have sufficed. This is called the *nack implosion* problem. As shown in Fig. 6, this waste can be reduced if a destination suppresses its own negative acknowledgment if it has received a negative acknowledgment that some other destination transmitted (12,13). Suppression of negative acknowledgments is combined with a carefully chosen delay before the negative acknowledgment is transmitted to minimize the probability that multiple negative acknowledgments are transmitted.

The second problem with negative acknowledgments is that they provide no indication to the source that all, or even any, destinations have received the messages. To ensure that the source can retransmit any message for which it might receive a negative acknowledgment, the source would need to retain every message indefinitely. Consequently, negative acknowledgments are typically used in combination with positive acknowledgments. The positive acknowledgments confirm that messages have been received by every destination and will not subsequently need to be retransmitted, thus allowing the source to recover the buffer space used by those messages.

The use of acknowledgments and retransmissions is ineffective for synchronous systems because it introduces arbitrary delays into the delivery of messages, delays that might exceed the specified bounds. Consequently, in many synchronous designs, processes transmit messages multiple times, typically over multiple communication paths and possibly multiple times on each path. With a proper design and a high-quality communication medium, the probability that no copy of the message reaches the destination is negligible (14).

Causal Order Algorithms

To determine causal dependencies between messages and delivery of messages in causal order, additional information must be included in the messages to indicate their causal predecessors. A naive strategy would require a process to include in every message it transmits a list of all messages it has received since the previous message it transmitted.

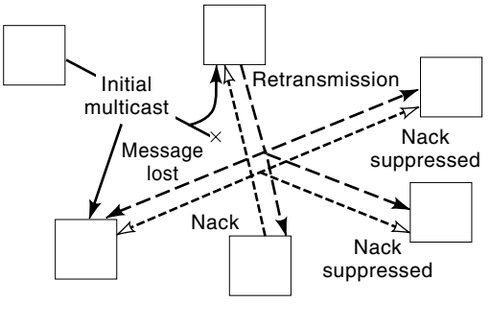

Figure 6. Excessive numbers of negative acknowledgments and retransmissions can be avoided if each process suppresses its negative acknowledgment or retransmission on receiving a similar transmission from another process.

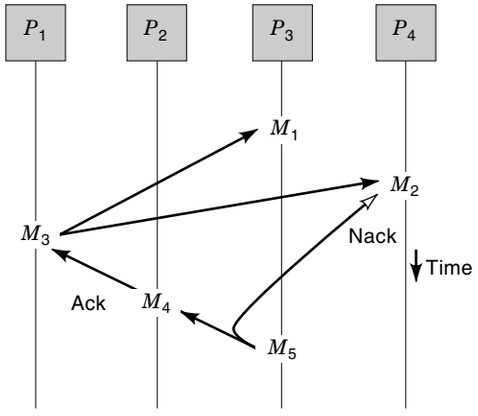

Figure 7. The transitivity of acknowledgments, piggybacked on regular messages, can be used to derive a causal order while requiring little additional information to be transmitted.

A more sophisticated and efficient algorithm exploits transitivity of positive acknowledgments (15,16). As shown in Fig. 7, message M_3 transmitted by process P_1 contains positive acknowledgments of messages M_1 and M_2. If process P_2 now transmits message M_4 containing a positive acknowledgment of M_3, P_2's message also implicitly acknowledges messages M_1 and M_2 and indicates that M_4 causally follows M_1, M_2 and M_3. If process P_3 has received messages M_1, M_3 and M_4 but has not received message M_2, then P_3 transmits message M_5 containing a positive acknowledgment of M_4 and a negative acknowledgment of M_2. The positive acknowledgment of M_4 implicitly acknowledges M_1 and M_3 and indicates that M_5 causally follows M_1, M_3 and M_4. The negative acknowledgment of M_2 serves to trigger a retransmission of M_2 so that M_2 can be delivered before M_5. Because maintaining the graph structure used by this strategy to determine the causal dependencies is computationally expensive, a variation (17) on this strategy requires a process to receive all of the predecessors of a message before it issues a positive acknowledgment of the message. Thus, the positive acknowledgments directly yield the causal dependencies, whereas the negative acknowledgments trigger retransmissions. This reduces the computational cost of deriving the causal dependencies, with a small cost in increased latency.

Another strategy commonly used to determine a causal order on messages exploits a vector clock (2). Each process maintains a local clock that can be either a real-time clock or a logical Lamport clock (9). A process maintains a logical Lamport clock as follows. When the process receives a message, it compares its local clock with the timestamp in the message. If the value of the message timestamp is greater than the value of its local clock, the process advances its local clock to match the timestamp. When the process transmits a message, it first increments its local clock and then uses that value to timestamp the message.

As shown in Fig. 8, each process also maintains a local vector clock that contains one entry in the vector for each process in the group. The process's own entry in the vector is its own Lamport clock. When a process transmits a message, it includes the vector clock in the message as a vector timestamp. When a process receives a message, it compares every entry in the message's vector timestamp with the corresponding entry in its local vector clock. If a value in the message's vector timestamp is greater than the corresponding value in its local clock, the process advances the entry in its local clock to the corresponding value in the message.

To determine the causal order between two messages, a process compares the corresponding entries in the vector timestamps of the messages. If every entry in one message's vector timestamp is greater than or equal to the corresponding entry in the other message's vector timestamp, then that message causally follows the other message. If both vector timestamps contain an entry that is greater than the corresponding entry in the other message's vector timestamp, then the two messages are concurrent and neither follows the other.

The vector clock strategy is effective only if the number of processes is small. As the number of processes increases, the transmission cost for the vector timestamp, and the computational cost of maintaining the vector clock, increase proportionately.

Some group communication systems are based entirely on a causal order on the messages (18). Other group communication systems do not construct a causal order on the messages but rather impose a total order directly, where the total order satisfies the causal order requirement. In general, such total order algorithms are as efficient as causally ordered algorithms within a local area but incur higher latency over wide areas.

Most synchronous systems do not construct explicit causal or total orders on messages during system operation. Rather, any causal or total order dependencies are considered in advance during the design of the system and the development of the preplanned schedule of operations (14).

Total Order Algorithms

Total order algorithms can be classified as *symmetric* or *asymmetric,* depending on whether all processes play the same role

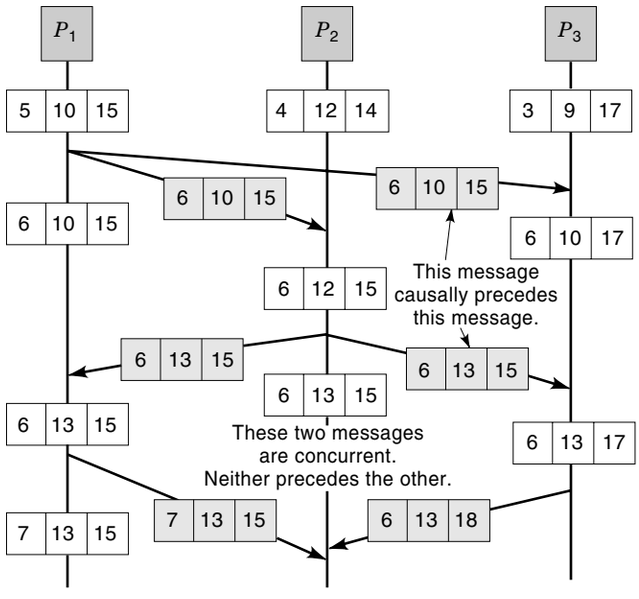

Figure 8. Vector clocks, maintained by the processes and included in each message, allow the causal order to be derived.

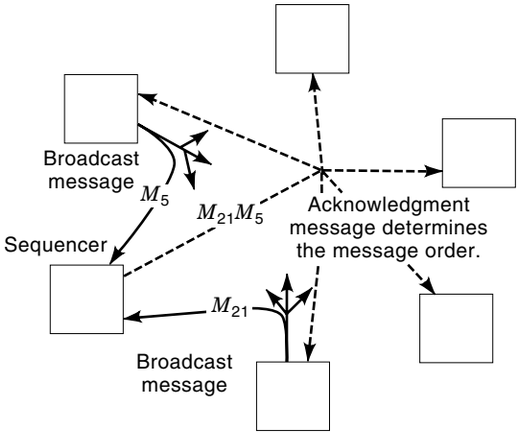

Figure 9. Acknowledgment messages, broadcast by the sequencer process, impose the total order on messages broadcast by the other processes.

or some processes are distinguished from others. Typical asymmetric algorithms are sequencer algorithms in which one process determines the ordering of messages broadcast by the other processes, and also token algorithms in which a process can broadcast only when it holds a token that rotates through the set of processes. Asymmetric algorithms are quite efficient, but they are vulnerable to a single point of failure.

Algorithms based on timestamping messages are more symmetric and are highly efficient, but they may exhibit high latency in wide-area networks. Intermediate between the symmetric and asymmetric algorithms are the hybrid algorithms in which a central core of processes executes a symmetric total order algorithm and other processes transmit their own messages to one of the core processes for ordering and broadcasting.

None of the preceding algorithms is fault-tolerant. When a process becomes faulty, the total order algorithm blocks temporarily until the membership algorithm has detected and diagnosed the fault and has formed a new membership that excludes the faulty process. A completely different class of algorithms contains fault-tolerant total ordering algorithms, based on voting (15,19). Such algorithms continue to order messages even though some processes are faulty. They are, however, quite sophisticated and computationally expensive and, thus, have not been widely used.

Sequencer Algorithms. In the sequencer algorithms, one process is responsible for determining the total order on messages broadcast by all processes. In the Amoeba system (20), every process transmits its messages over a point-to-point connection to the sequencer process. The sequencer then determines the message order and broadcasts the messages. In alternative sequencer algorithms (21–24), shown in Fig. 9, the originators of the messages broadcast their messages. The messages are received by the sequencer, which then determines the total order and broadcasts an acknowledgment message that lists the various broadcast messages in the total order. Other processes cannot deliver a message until they have received both the message and the acknowledgment message from the sequencer containing the ordering information. If a process does not find the message it broadcast listed in the acknowledgment message, it rebroadcasts the message. If a process finds a message listed in the acknowledgment message but has not received the message, it requests a retransmission with a negative acknowledgment.

If the sequencer broadcasts the messages, each message is transmitted twice, which increases the load on the communication medium, and the sequencer may become a bottleneck. If the originator broadcasts its own messages, the load on the communication medium and on the sequencer is reduced, but the processes then receive two transmissions, the broadcast message and the acknowledgment message from the sequencer, which increases the load on the processes.

Because the sequencer is a single point of failure and a processing bottleneck, and also to avoid the need for positive acknowledgments, most sequencer algorithms rotate the responsibility for sequencing through the processes in the group. In Ref. (21), the acknowledgment message not only orders the message but also transfers the responsibility for sequencing the next batch of messages to the next process. If the sequencer process fails, this rotation stops, as does the delivery of messages, until the membership algorithm has removed the faulty process from the membership.

Token Algorithms

Another strategy for totally ordering messages exploits a token rotating around a logical ring (3,25–27). Only the holder of the token can broadcast messages. The token contains a sequence number that is incremented every time a message is broadcast, which imposes a total order on the messages broadcast in the group, as shown in Fig. 10. The token also contains additional information, including positive and negative acknowledgments and also flow control information. To avoid the overhead of circulating the token when there are no messages to broadcast, the algorithm may contain a mechanism for stopping and restarting the circulation of the token.

Token algorithms can be very efficient for small groups with a heavy load of message communication. The processing required is simple and the flow control information in the token is effective for ensuring that the buffers at the destinations do not overflow, even under high load. For large groups under low load, the delay waiting for the token to arrive causes a higher latency than for a sequencer algorithm. This

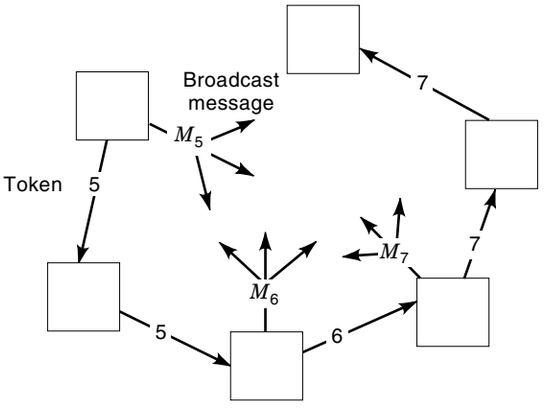

Figure 10. In a token algorithm, the token contains a sequence number that is incremented for every message broadcast, imposing a total order on the messages broadcast in the group.

latency can be minimized by passing the token only to the processes that have messages to broadcast and that have requested the token (4,28). If, however, the token does not visit all processes, alternative arrangements must be made for collecting the positive acknowledgments that must be obtained from all of the processes in order to manage buffer space efficiently.

A faulty process causes loss of the token and stops message transmission until a membership algorithm has removed the faulty process from the membership. On the other hand, the continuously circulating token allows rapid detection of faulty processes.

Timestamp Algorithms. An elegant strategy for totally ordering messages involves timestamping the messages and delivering them in timestamp order (3,29). The timestamps can be derived from either a logical Lamport clock or, alternatively, from synchronized physical clocks. In order that a process can deliver the message with the lowest timestamp, it must know that it will not subsequently receive a message with a lower timestamp from any other process in the group. This can be guaranteed if it receives the messages in reliable FIFO order and if it has received a message with a higher timestamp from every other process in the group. Some processes may need to transmit null messages to ensure that a message from them is always available, to allow messages from other processes to be delivered promptly.

Timestamp algorithms involve simple program code and, consequently, can be very efficient. They also have the advantage that messages can be ordered within small groups with the confidence that the local total and causal order is consistent with a system-wide total and causal order, precluding subtle ordering anomalies. The disadvantage of timestamp algorithms, particularly in large groups where many processes transmit infrequently, is that large numbers of null messages may be required. Algorithms have been devised to combine null messages from many processes, thereby reducing their number. As in the sequencer and token algorithms, a faulty process causes message ordering to stop until the membership algorithm has removed the faulty process from the membership.

Hybrid Algorithms. Hybrid algorithms (30,31) for total ordering messages provide efficient operation in large systems where many processes have messages to transmit only occasionally. Certain processes, typically those with high transmission rates and also high bandwidth communication links, are designated to be core processes, as shown in Fig. 11. The core processes broadcast and deliver messages using one of the other total ordering algorithms. Other processes transmit their messages, point-to-point, to any core process, which then orders and broadcasts those messages.

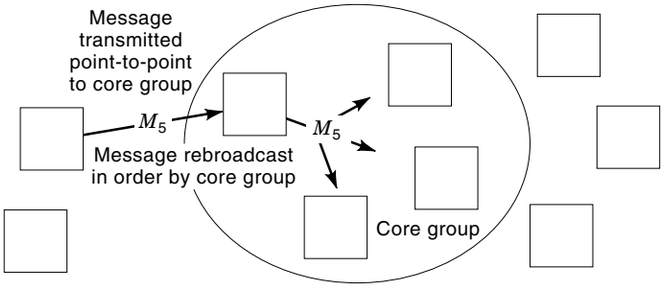

Figure 11. In a hybrid message-ordering algorithm, the core processes order and broadcast messages sent to them by the other processes.

Effective operation of a hybrid algorithm depends on an appropriate choice of processes for the core. Algorithms have been developed to determine that choice dynamically, adding processes to the core, or removing them, as their message transmission rates change.

Hybrid algorithms are particularly important when the group size is large (thousands of processes) but only a few processes transmit frequently, as may occur in Internet applications. If, however, the listen-only processes require reliable delivery of messages, they must still transmit positive and negative acknowledgments, and care is required to avoid ack implosion (13).

Voting Algorithms. The voting algorithms that produce a total order on messages (15,19) are completely different from the algorithms described earlier. They start from a causal order derived from acknowledgments, as is shown in Fig. 7. Candidate messages that have not yet been ordered but do not follow any other unordered message are selected. Such messages are candidates for immediate advancement into the total order.

A voting strategy is used in which messages vote for messages that precede them in the causal order but that have not yet been advanced to the total order. If the causal order is narrow with few concurrent messages, so that it is almost a total order, the voting algorithm is likely to terminate in the first round. If the causal order is broad, several rounds of voting may be required, and termination of the voting algorithm depends on randomness properties of the causal order. Unfortunately, space does not permit a full description of the rather subtle voting algorithm or of the intricate proof of correctness (19).

The most interesting feature of the voting algorithms is that, unlike other total ordering algorithms, they are fault-tolerant and do not stop ordering messages in the presence of a faulty process. The absence of a hiatus in ordering messages is important for some applications. Moreover, unlike the other total ordering algorithms, the membership algorithm can be mounted above the total ordering algorithm (32), which allows the membership algorithm to be simpler and more robust. The disadvantage of the voting algorithm is its computational cost. The complexity of the algorithm is also a disadvantage because few developers want to use an algorithm if they do not understand why it works.

FLOW CONTROL ALGORITHMS

Group communication systems incur particularly severe flow control problems because, to achieve high performance, any one process must be able to transmit messages up to the capacity of the network and of the destinations. If, however, several processes transmit messages simultaneously at that rate, saturation of the communication medium can occur, resulting in message loss and retransmission. Moreover, several senders can transmit messages substantially faster than any destination can handle them. This causes messages to accumulate in the input buffers at the destinations until they

overflow and message loss occurs. In a local area, the high bandwidth of the communication medium may allow even a single sender to overwhelm the destinations. In a wide area, the critical resource is the available bandwidth of the network, which is often much lower than in a local area and is potentially highly variable because of contention with unrelated traffic. Experience demonstrates that message loss in modern communication networks is caused mainly by flow control and buffering problems.

The most effective flow control algorithms currently available for a local area are those used by token-based protocols (3,26). Only one process can broadcast or multicast at a time and the token carries flow control information from one process to the next around the ring. If the number of messages transmitted in one token rotation is restricted to the buffer capacity of the receivers, and if each process empties its buffer before releasing the token, buffer overflow is avoided. The token also carries information about the backlog of messages that could not be sent because of flow control, ensuring that all processes receive a fair share of the medium.

Sequencer and timestamp algorithms use a window flow control strategy in the style of that used by TCP/IP (33,34). When a process broadcasts a message, it reduces the remaining window space, restoring the window space when it has received acknowledgments for that message from all members of the group (and, thus, no longer needs to buffer the message for possible retransmission). If each process in a group is provided with its own window then, given finite resources, those windows must be smaller than what would have been possible had the processes shared a window. Thus, the transmission rate of a process will be restricted because some of the resources have been allocated for other processes. If all processes share a window, then a process must reduce the space in the window for each message it receives as well as for each message it transmits, again restoring the window space when it has received acknowledgments from all members of the group. However, with a shared window and without control over multiple concurrent transmissions, several processes may transmit messages that attempt to utilize the same residual window space, leading to buffer overflow and message loss. For wide-area group communication systems operating over the Internet, window flow control is essential to achieve good performance. Internet switches use a flow control strategy, Random Early Drop (RED) (35), closely matched to TCP/IP. In contrast, wide-area group communication systems operating over ATM must accommodate the rate-based quality of service mechanisms of ATM (34), defined for each transmitter separately. In both cases, the relatively long delay until acknowledgments are received, which is inevitable in wide-area networks, can severely degrade the performance.

FAULT DETECTORS

A *fault detector* is a distributed algorithm such that each process has a local fault detector module that reports the processes it currently suspects as being faulty (1).

For fail-stop and crash faults, fault detectors are typically based on timeouts that are local to the process, with no communication between processes. If a process has not received a message from another process within a certain period of time, its fault detector adds that process to the list of those suspected of being faulty. This includes processes that have not acknowledged receipt of a message within a reasonable amount of time. Failure to acknowledge receipt of a message forces other processes to retain the message in their buffers for possible retransmission and could exhaust that buffer space, causing the system to stop. For Byzantine faults, fault detectors must rely on costly techniques such as reliable broadcast or diffusion algorithms and message signatures.

Even in models that admit only fail-stop and crash faults, fault detectors are inherently unreliable because processes that are nonfaulty but excessively slow or processes that fail to receive a message an excessive number of times may be suspected, whereas processes that are faulty may not be suspected immediately.

MEMBERSHIP ALGORITHMS

The two types of membership, primary component membership and partitionable membership, shown in Fig. 3, satisfy different application objectives. A primary component membership is most useful when the application must maintain a single consistent state for its data in the primary component, at the cost of suspending the operation of processes in the nonprimary components, for example, in banking. A partitionable membership is appropriate when all processes must continue operation, with the cost of reconciling inconsistent data when communication is reestablished between disconnected components, for example, in industrial control.

Algorithms exist for both types of membership (10,26,32,36–39), and significant problems exist for both (40,41). For primary component membership, it is possible that no membership satisfies the requirements for being the primary membership (such as a majority of the processes in the group). In practice, however, membership algorithms almost always find primary components quite quickly. For partitionable membership, the algorithm may form a trivial or inappropriate membership, such as allowing every process to form an isolated singleton membership. In practice, however, partitionable membership algorithms do not choose such memberships in preference to other more appropriate memberships.

Robust membership algorithms are difficult to program because they must operate under uncertain conditions and must handle additional faults that occur during their operation. Implementation details, such as the relative lengths of timeouts, are very important for robust operation and depend on the underlying platform on which the algorithms operate. We provide next a broad outline of the strategies used by typical membership algorithms. More details can be found in Refs. 26 and 36.

Typical membership algorithms involve four phases—initiation, discovery, agreement, and recovery—as shown in Fig. 12. *Initiation* of the membership algorithm may result from an explicit request by a process to join or leave the group, a suspicion by a fault detector, or reception of a message from a foreign process (not in this membership but in a concurrent membership within a partitioned system) after remerging of a partitioned system.

In the *discovery* phase, all processes broadcast messages inviting responses from other processes. Each such process broadcasts responses that enumerate all processes from

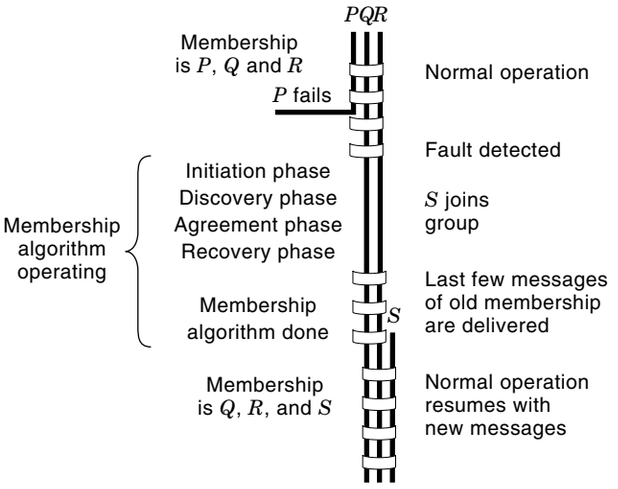

Figure 12. The four phases of a membership algorithm.

which they have received messages, the *known* set, and all processes that they suspect as having failed, the *fail* set. On receipt of such a message, a process merges all of the processes in the known set of the message into its own known set. Similarly, a process merges all of the processes in the fail set of the message into its own fail set. If a process has not received a response from a process in its known set within a timeout, it also adds that process to its fail set.

The discovery phase ends either by a timeout or by agreement on a membership. The discovery phase can, however, be reentered at any time if further processes are suspected, if agreement cannot be reached, or if one or more processes do not install the new membership.

The *agreement* phase seeks to find a set of processes such that every process in that set agrees that its proposed new membership is that set. The proposed membership is typically the difference of the known set and the fail set. If the processes are only partially connected, so that some processes cannot communicate with other processes, a heuristic algorithm may be used to choose an appropriate membership. For a primary component membership, the proposed membership must also satisfy some size constraint or other criterion for being a primary component.

The agreement is then confirmed, typically by some variation of two-phase commit. The proposer, usually the process having the lowest identifier, broadcasts a proposal message. The other members then respond with a commit message. The proposer then broadcasts an install message to begin the recovery phase and install the new membership. If any process rejects the membership or does not respond, the proposer returns to the discovery phase. Similarly, if a process does not receive a propose or install message, it returns to the discovery phase.

In the recovery phase, the processes first complete the delivery of messages from the old membership and then install the new membership. The processes in the new membership and the same old membership exchange information regarding messages that they have received from the old membership and then retransmit those messages so that all members have them. The messages are then ordered and delivered to ensure virtual synchrony. When all messages of the old membership are delivered, the algorithm delivers a membership change message announcing the membership change, enumerating the new membership, and starting normal operation with the new membership. Even before the delivery of some of the messages of the old membership, the membership algorithm may have delivered additional membership change messages reporting the loss of processes. Such additional messages are necessary to achieve extended virtual synchrony. If a process determines that any member of its new membership has returned to the discovery phase, it first completes its installation of the new membership and then reinvokes the membership algorithm.

For a primary component membership algorithm, it is essential that only a single sequence of memberships exists over time. If the proposer becomes faulty at a critical moment, it may be impossible for the remaining processes to determine whether the proposer has installed the new membership. The system must then stop until the proposer has recovered. This risk of a hiatus can be reduced, but not eliminated, by using three-phase commit in place of two-phase commit.

For a partitionable membership algorithm, termination is easy to demonstrate. The known set and the fail set are monotonically increasing and bounded above by the finite number of potential members. Each attempt to form a membership can be defeated by a process that was previously unknown, causing an increase in the known set, or by a process that does not respond, causing an increase in the fail set. Because both sets increase monotonically and are bounded above by the set of potential members, the algorithm terminates, possibly in a singleton membership containing only the process itself.

Membership algorithms for operation on top of a fault-tolerant total ordering algorithm (32) are often simpler and more elegant than the algorithm outlined earlier. This simplicity comes at the expense of greater complexity in the fault-tolerant total ordering algorithm.

For synchronous systems, membership algorithms are much simpler than for asynchronous systems (14,42). Typically, at the end of each prescheduled sequence of message exchanges, each process reports the set of processes from which it received messages during that sequence. This set of processes constitutes its proposed membership. If a process receives a membership that differs from its own, it can choose either to exclude that process from its membership or to exclude other processes from its membership so as to bring its membership into agreement with that of the other process. In principle, these choices are heuristic choices for synchronous systems as they are for asynchronous systems. Practical synchronous systems, however, typically have simpler and more robust communication media and, thus, incur fewer problems in reaching agreement on a membership quickly.

FUTURE DIRECTIONS

Much research remains to be undertaken in the area of fault-tolerant distributed systems. An important research topic is the integration of group communication protocols with protocols for real-time, multimedia, and data transfer. Real-time protocols, such as are used for instrumentation and control, typically seek to provide low latency and low jitter (variance in latency) but not reliable delivery because new data are

transmitted more or less continuously. Multimedia protocols that provide broadcasting or multicasting of audio and video, need low latency and low jitter but not reliable delivery. Data transfer protocols provide reliable delivery and may provide broadcasting or multicasting but usually do not need message ordering between multiple sources. The use of these protocols in a distributed system depends on group communication for overall coordination. It is essential to establish a causal order between the control information transmitted through the group communication protocol and the start or end of real-time, multimedia, or data transmission.

As group communication protocols become more established, they will be used in larger systems and over wider areas. Over wide areas, with high data rates, existing flow control algorithms are ineffective. To preclude overwhelming the buffers in the intermediate switches, a relatively small window is needed, but messages in the window are quickly transmitted and the source then remains idle for a long time until the acknowledgments return. New flow control algorithms will be required. With existing protocols, the latency to message ordering and delivery can increase substantially over a wide area. Some increase in latency is inevitable because of the propagation delay through the network, but new protocols that can order messages with a latency that is close to this minimum will be required (27).

Wide-area systems are also subject to network partitioning; however, many applications require all components of a partitioned system to continue operation. Existing group communication systems provide message delivery and membership algorithms that continue to operate in all components of a partitioned system. Even though the system is partitioned, the disconnected components can perform operations that are inconsistent with those performed in other components. When communication is eventually restored, these inconsistencies must be reconciled. The programming required to achieve such reconciliation is currently quite difficult, and expensive manual intervention may be required. Proposals have been made (44–46) to simplify this programming, although human insight is still required to establish the application requirements on which such programming depends. The development of strategies for preventing or reconciling inconsistencies in partitioned systems is an important topic of research.

Group communication is in the middle of a range of approaches to the development of fault-tolerant distributed systems. One end of that range is focused on efficiency, while the other end is focused on simplification of the application programming. When communication networks and group communication protocols were slow, a strong emphasis on efficiency was appropriate (47). A similar concern has led to the development of microprotocol toolkits (4) from which a custom group communication protocol can be constructed, optimized specifically for the particular application. With increasing network performance and more efficient protocols, some of that efficiency can be sacrificed for simpler application programming.

The group communication protocols described in this article employ a message-passing application programmer interface. This message-passing interface necessarily exposes to the application programmer the problems of distribution, replication, consistency, and fault tolerance. Correct solutions to these problems require considerable skill and experience, and typical application programmers are not well-trained to solve those problems. Consequently, fault-tolerant distributed systems are still quite difficult and expensive to program.

New approaches to building fault-tolerant distributed systems are being investigated. Using the Common Object Request Broker Architecture (CORBA) (48,49), such systems (46,50) provide transparent object replication and fault tolerance. This allows the application programmer to write a distributed object program as though it were to operate unreplicated, without affecting the application programming or the functional behavior of the application. The approach still employs group communication protocols such as those described here, but does not expose those protocols to the application programmer. Such an approach will make the benefits of fault-tolerant distributed systems available to a wider range of applications.

BIBLIOGRAPHY

1. T. D. Chandra and S. Toueg, Unreliable failure detectors for reliable distributed systems, *J. ACM,* **43** (2): 225–267, 1996.
2. K. P. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit,* Los Alamitos, CA: IEEE Comput. Soc. Press, 1994.
3. L. E. Moser et al., Totem: A fault-tolerant multicast group communication system, *Commun. ACM,* **39** (4): 54–63, 1996.
4. R. van Renesse, K. P. Birman, and S. Maffeis, Horus: A flexible group communication system, *Commun. ACM,* **39** (4): 76–83, 1996.
5. F. Cristian, Synchronous and asynchronous group communication, *Commun. ACM,* **39** (4): 88–97, 1996.
6. M. J. Fischer, N. A. Lynch, and M. S. Paterson, Impossibility of distributed consensus with one faulty process, *J. ACM,* **32** (2): 374–382, 1985.
7. M. Ben-Or, Randomized agreement protocols, in B. Simons and A. Spector (ed.), *Fault-Tolerant Distributed Computing,* Berlin, Germany: Springer-Verlag, 1990, pp. 72–83.
8. G. Bracha and S. Toueg, Asynchronous consensus and broadcast protocols, *J. ACM,* **32** (4): 824–840, Oct. 1985.
9. L. Lamport, Time, clocks, and the ordering of events in a distributed system, *Commun. ACM,* **21** (7): 558–565, 1978.
10. D. Dolev, D. Malki, and R. Strong, *A framework for partitionable membership service,* Tech. Rep. CS95-4, Inst. Comput. Sci., Hebrew Univ., Jerusalem, Israel, 1995.
11. L. E. Moser et al., Extended virtual synchrony, *Proc. 14th IEEE Int. Conf. Distrib. Comput. Syst.,* Poznan, Poland, 1994, pp. 56–65.
12. S. Floyd et al., A reliable multicast framework for light-weight sessions and application level framing, *ACM / IEEE Trans. Netw.,* **5**: 784–803, 1997.
13. K. Berket, L. E. Moser, and P. M. Melliar-Smith, The InterGroup protocols: Scalable group communication for the Internet, IEEE GLOBECOM '98: *3rd Global Internet Mini-Conf.,* Sydney, Australia, 1998.
14. H. Kopetz and G. Grunsteidl, TTP—A protocol for fault-tolerant real-time systems, *IEEE Comput.,* **27** (1): 14–23, 1994.
15. P. M. Melliar-Smith, L. E. Moser, and V. Agrawala, Broadcast protocols for distributed systems, *IEEE Trans. Parallel Distrib. Syst.,* **1**: 17–25, 1990.
16. P. M. Melliar-Smith and L. E. Moser, Trans: A reliable broadcast protocol, *IEE Proc. I Trans. Commun.,* **140**: 481–492, 1993.

17. Y. Amir et al., Transis: A communication sub-system for high availability, *Proc. 22nd IEEE Int. Symp. Fault-Tolerant Comput.,* Boston, MA, 1992, pp. 76–84.
18. S. Mishra, L. L. Peterson, and R. D. Schlichting, Consul: A communication substrate for fault-tolerant distributed programs, *Distrib. Syst. Eng.,* **1** (2): 87–103, Dec. 1993.
19. L. E. Moser, P. M. Melliar-Smith, and V. Agrawala, Asynchronous fault-tolerant total ordering algorithms, *SIAM J. Comput.,* **22** (4): 727–750, 1993.
20. M. F. Kaashoek and A. S. Tanenbaum, Group communication in the Amoeba distributed operating system, *Proc. 11th IEEE Int. Conf. Distrib. Comput. Syst.,* Arlington, TX, 1991, pp. 222–230.
21. J. M. Chang and N. F. Maxemchuk, Reliable broadcast protocols, *ACM Trans. Comput. Syst.,* **2**: 251–273, 1984.
22. F. Cristian and S. Mishra, The pinwheel asynchronous atomic broadcast protocols, *Proc. 2nd Int. Symp. Autonomous Decentralized Syst.,* Phoenix, AZ: Apr. 1995, pp. 215–221.
23. W. Jia, J. Kaiser, and E. Nett, RMP: Fault-tolerant group communication, *IEEE Micro,* **16** (2): 59–67, Apr. 1996.
24. B. Whetten and S. Kaplan, A high performance totally ordered multicast protocol, *Proc. Int. Workshop Theory and Practice Distrib. Syst.,* Dagstuhl Castle, Berlin, Germany: Springer-Verlag, Sept. 1994, pp. 33–57.
25. T. Abdelzaher et al., RTCAST: Lightweight multicast for real-time process groups, *Proc. 1996 IEEE Real-Time Technology and Applications Symp.,* Brookline, MA: June 1996, pp. 250–259.
26. Y. Amir et al., The Totem single-ring ordering and membership protocol, *ACM Trans. Comput. Syst.,* **13**: 311–342, 1995.
27. B. Rajagopalan and P. K. McKinley, A token-based protocol for reliable, ordered multi-cast communication, *Proc. 8th IEEE Symp. Reliable Distrib. Syst.,* Seattle, WA: Oct. 1989, pp. 84–93.
28. G. A. Alvarez, F. Cristian, and S. Mishra, On-demand asynchronous atomic broadcast, *Proc. 5th IFIP Int. Working Conf. Dependable Computing for Critical Applications,* Urbana-Champaign, IL: 1995, pp. 119–137.
29. D. A. Agarwal et al., The Totem multiple-ring ordering and topology maintenance protocol, *ACM Trans. Comput. Syst.,* **16**: 93–132, 1998.
30. P. D. Ezhilchelvan, R. A. Macedo, and S. K. Shrivastava, Newtop: A fault-tolerant group communication protocol, *Proc. 15th Int. Conf. Distrib. Computing Syst.,* Vancouver, BC, Canada: May/June 1995, pp. 296–306.
31. L. E. T. Rodrigues, H. Fonseca, and P. Verissimo, Totally ordered multicast in large-scale systems, *Proc. 16th IEEE Int. Conf. Distrib. Comput. Syst.,* Hong Kong, 1996, pp. 503–510.
32. L. E. Moser, P. M. Melliar-Smith, and V. Agrawala, Processor membership in asynchronous distributed systems, *IEEE Trans. Parallel Distrib. Syst.,* **5**: 459–473, 1994.
33. D. E. Comer, *Internetworking with TCP/IP,* Englewood Cliffs, NJ: Prentice-Hall, 1995.
34. S. Floyd and V. Jacobson, Random early detection gateways for congestion avoidance, *IEEE/ACM Trans. Netw.,* **1**: 397–413, 1993.
35. Y. Amir et al., Membership algorithms for multicast communication groups, *Proc. 6th Int. Workshop Distrib. Algorithms,* Haifa, Israel, 1992, pp. 292–312.
36. F. Cristian, Reaching agreement on processor-group membership in synchronous distributed systems, *Distrib. Comput.,* **4** (4): 175–187, 1991.
37. M. A. Hiltunen and R. D. Schlichting, A configurable membership service, *IEEE Trans. Comput.* **47** (5): 573–586, May 1998.
38. F. Jahanian, S. Fakhouri, and R. Rajkumar, Processor group membership protocols: Specification, design and implementation, *Proc. 12th Symp. Reliable Distrib. Syst.,* Princeton, NJ: Oct. 1993, pp. 2–11.
39. A. M. Ricciardi and K. P. Birman, *Process membership in asynchronous environments,* TR 93-1328, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1993.
40. E. Anceaume et al., *On the formal specification of group membership services,* Tech. Rep. 95-1534, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1995.
41. T. D. Chandra et al., *On the impossibility of group membership,* Tech. Rep. 95-1548, Dept. of Computer Science, Cornell Univ., Ithaca, NY, 1995.
42. A. S. Tanenbaum, *Computer Networks,* Upper Saddle River, NJ: Prentice-Hall, 1996.
43. R. Koch, L. E. Moser, and P. M. Melliar-Smith, *Global causal ordering with minimal latency,* Tech. Rep. 98-08, Dept. Electr. Comput. Eng., Univ. California, Santa Barbara, 1998.
44. O. Babaoglu, A. Bartoli, and G. Dini, Enriched view synchrony: A programming paradigm for partitionable asynchronous distributed systems, *IEEE Trans. Comput.,* **46**: 642–658, 1997.
45. P. M. Melliar-Smith and L. E. Moser, Surviving network partitioning, *IEEE Comput.,* **31** (3): 62–69, 1998.
46. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, Replica consistency of CORBA objects in partitionable distributed systems, *Distrib. Syst. Eng.,* **4**: 139–150, 1997.
47. D. R. Cheriton and D. Skeen, Understanding the limitations of causally and totally ordered communication, *Proc. 14th ACM Symp. Operating Systems Principles,* Asheville, NC: Dec. 1993; *Operating Syst. Rev.,* **27** (5): 44–57, Dec. 1993.
48. Object Management Group. *The Common Object Request Broker: Architecture and Specification,* Rev. 2.1, OMG Tech. Doc. PTC/97-09-01, 1997.
49. R. M. Soley, *Object Management Architecture Guide,* Object Management Group, OMG Tech. Doc. 92-11-1, 1992.
50. L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan, Consistent object replication in the Eternal system, *Theory Practice Object Syst.,* **4** (2): 81–92, 1998.

P. M. MELLIAR-SMITH
L. E. MOSER
University of California