Software-Based Replication for Fault Tolerance

Replication handled by software on off-the-shelf hardware costs less than using specialized hardware. Although an intuitive concept, replication requires sophisticated techniques for successful implementation. Group communication provides an adequate framework.

Rachid Guerraoui and André Schiper
Swiss Federal Institute of Technology

One solution for achieving fault tolerance is to build software on top of specialized hardware. Companies such as Tandem and Stratus have successfully pursued this solution for some applications. Economic factors, however, motivate the search for cheaper software-based fault tolerance; that is, replication handled entirely by software on off-the-shelf hardware. Although replication is an intuitive, readily understood idea, its implementation is difficult. Here we present a survey of the techniques developed since the mid-80s to implement replicated services, emphasizing the relationship between replication techniques and group communication.

PROCESSES AND COMMUNICATION LINKS

A typical distributed system consists of a set of processes exchanging messages via communication links. Processes can be correct or incorrect. Correct processes behave according to their specifications, while incorrect processes either crash (stop receiving or sending messages) or behave maliciously (send messages that do not follow the specification). In this article, we consider only crash failures.

There are two basic system models for communication links: synchronous and asynchronous. The synchronous model assumes that a known value bounds the transmission delay of messages. In contrast, the asynchronous model does not set any limit on the transmission delay, which adequately models an unpredictable load of processors and links. This makes the asynchronous model more general, and we consider only this model here.

CORRECTNESS CRITERION

To discuss replicated servers, we must first explain the correctness criterion *linearizability*.[1] Sometimes called one-copy equivalence, linearizability gives clients the illusion of nonreplicated servers, a highly desirable trait because it preserves the program's semantics.

Suppose a client process p_i invokes operation op with arguments *arg* on a nonreplicated server x. This is denoted as op(*arg*). After client p_i invokes op, server x sends a response of the form ok(*res*), where ok denotes successful completion of the invocation and *res* the invocation response (*res* can be empty). Fault tolerance is achieved by replicating the servers on different sites of the distributed system. We denote server x's n replicas by $x^1, ..., x^n$.

Replication does not change the way we denote invocations and the corresponding responses. Client p_i still issues one invocation, op(*arg*), and considers one response, ok(*res*), as Figure 1 shows. However, if the set of replicas x^i do not handle the invocation properly, the response can be incorrect. For example, suppose server x implements a FIFO queue using operations enqueue() and dequeue(). Assume an initially empty queue and client processes p_i and p_j that perform two invocations:

- Process p_i calls enqueue(*a*) and then dequeue(), which returns response ok(*b*).
- Process p_j calls enqueue(*b*) and then dequeue(), which returns response ok(*b*).

Obviously this execution is incorrect; *b* has been enqueued once but dequeued twice. Replication, however, makes such an erroneous execution possible. Suppose server x is replicated twice (x^1 and x^2) and the following happens:

- Replica x^1 first receives the enqueue(*a*) invocation from p_i then the enqueue(*b*) invocation

0018-9162/97/$10.00 © 1997 IEEE

from p_j. The dequeue() invocation from p_i causes response ok(a) to be sent to p_i. Finally, the dequeue() invocation from p_j causes response ok(b) to be sent to p_j.
- Replica x^2 receives the enqueue(b) invocation from p_j, then the enqueue(a) invocation from p_i. The dequeue() invocation from p_i leads to ok(b). Finally, the dequeue() invocation from p_j leads to ok(a).

If p_i selects ok(b) from x^2 (because it receives this response first), and p_j selects ok(b) from x^1 the result is the incorrect execution that appears to enqueue b once and dequeue it twice.

We cannot linearize this execution because replicas x^1 and x^2 receive and handle invocations enqueue(a) from p_i and enqueue(b) from p_j in a different order. To ensure linearizability, client invocations must satisfy the following properties:

- *Order.* Given invocations op(arg) by client p_i and op′ (arg′) by client p_j on replicated server x, if x^k and x^l handle both invocations, they handle them in the same order.
- *Atomicity.* Given invocation op(arg) by client p_i on replicated server x, if one replica of x handles the invocation, then every correct (noncrashed) replica of x also handles op(arg).

REPLICATION TECHNIQUES

Two fundamental classes of replication techniques ensure linearizability: primary-backup and active. Due to space constraints, we do not cover hybrid techniques that combine these techniques.

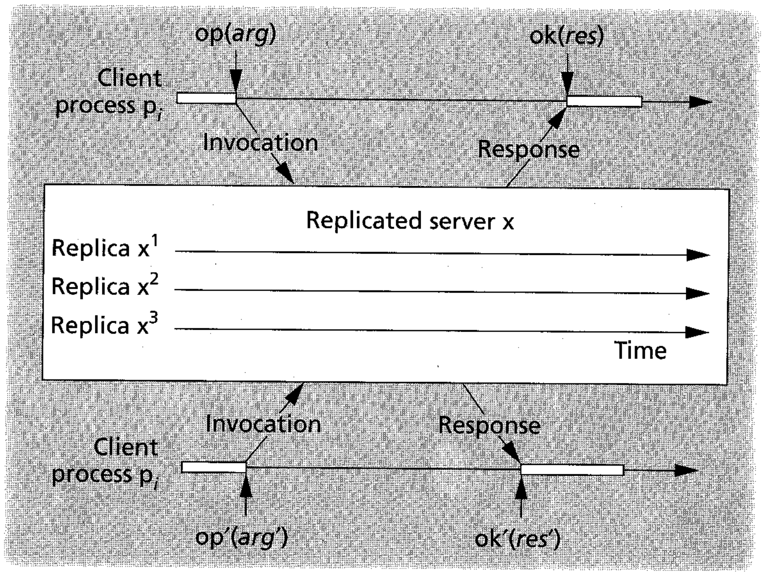

Figure 1. Interaction with a replicated server.

Primary-backup replication

This technique[2] uses one replica, the primary, that plays a special role: it receives invocations from client processes and returns responses. Server x's primary replica is denoted prim(x); other replicas are backups. Backups interact directly only with the primary replica, not the client processes. Figure 2 illustrates how the primary replica handles the invocation of op(arg) by client p_i (assuming the primary replica does not crash).

- Process p_i sends op(arg) to prim(x) (primary replica x^1) together with unique invocation identifier *invID*.
- Prim(x) invokes op(arg), which generates response *res*. Prim(x) updates its state and sends the update message (*invID,res,state-update*) to its backups. Value *invId* identifies the invocation, *res* the response, and *state-update*, the state update performed by the primary. Upon receiving

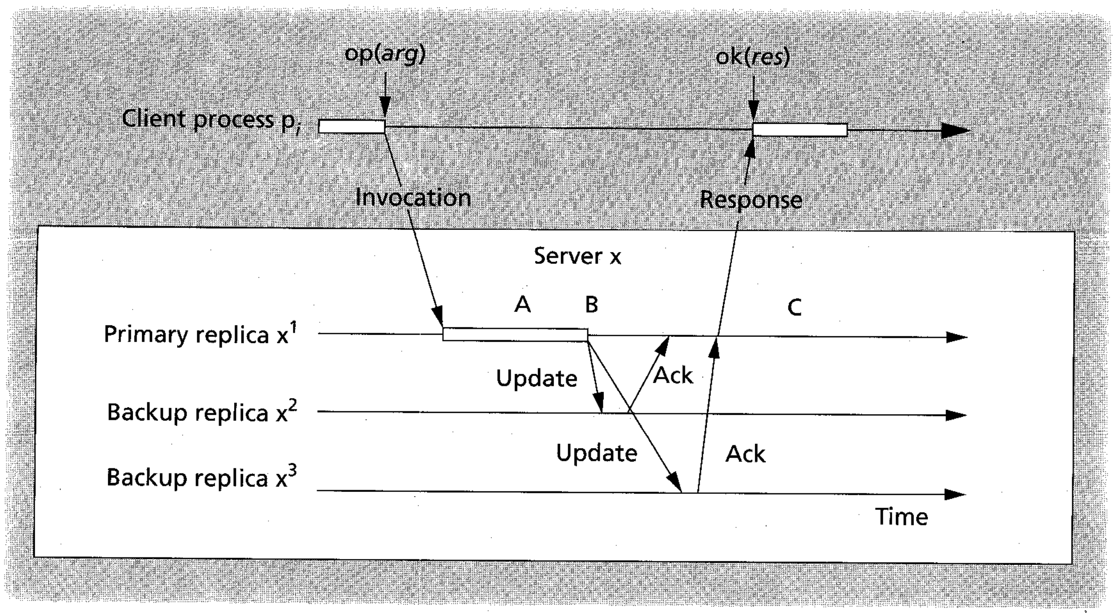

Figure 2. Primary-backup technique.

the update message, the backups update their state and return an acknowledgment to prim(x).
- Once the primary replica receives the acknowledgment from all correct (noncrashed) backups, it sends the response to p_i.

This scheme obviously ensures linearizability because the order in which the primary replica receives invocations defines the total order of all server invocations. The reception of the state-update message by all the backups ensures the atomicity property.

Ensuring linearizability despite the crash of the primary replica is more difficult. We can distinguish three cases, in which the primary replica crashes

- before sending the update message to the backups (point A in Figure 2)
- after (or while) sending the update message, but before the client receives the response (point B), or
- after the client receives the response (point C).

In all three cases, we must select a new primary replica.

In the third case, the crash is transparent to the client. In the first and second cases, the client will not receive a response to its invocation and will suspect a failure. After having learned the identity of the new primary replica, the client will reissue its invocation. In the first case, the new primary replica considers the invocation as new.

The second case is the most difficult to handle. The solution must ensure atomicity: either all or none of the backups must receive the update message. If none of the backups receive the message, the second case becomes similar to the first case. If all receive the update message, the operation of client process p_i updates the state of the backups, but the client does not receive a response and will reissue its invocation. The new primary replica needs *invID* and *res* to avoid handling the same invocation twice, which would produce an inconsistent state if the invocation is not idempotent. When the new primary replica receives invocation *invID*, it immediately returns response *res* to the client, rather than handling the invocation.

If we assume a perfect failure detection mechanism, the primary-backup replication technique is relatively easy to implement, apart from the atomicity issue just discussed. The implementation becomes much more complicated in an asynchronous system model because the failure detection mechanism is not reliable (that is, we have to handle the case in which a client incorrectly suspects the primary replica has crashed). The view-synchronous paradigm presented later defines the communication semantics that ensure correctness of the primary-backup technique despite an unreliable failure detection mechanism.

Active replication

Also called the state-machine approach,[3] this technique gives all replicas the same role without the centralized control of the primary-backup technique. Consider replicated server x and the invocation op(*arg*) issued by p_i. As shown in Figure 3,

- Invocation op(*arg*) goes to all the replicas of x.
- Each replica processes the invocation, updates its own state, and returns the response to client p_i.
- Client p_i waits until it receives the first response or a majority of identical responses. If the replicas do not behave maliciously, then p_i only waits for the first response.

Active replication requires noncrashed replicas to receive the invocations of client processes in the same order. This requires a communication primitive that satisfies the order and the atomicity properties introduced earlier. We discuss this primitive, called total-order multicast (also called atomic multicast), in detail later.

Comparing replication techniques

Active replication requires operations on the replicas to be deterministic, which is not the case in the primary-backup technique. *Determinism* means that an operation's outcome depends only on a replica's initial state and the sequence of operations it has already performed. Multithreaded servers typically lead to nondeterminism.

In active replication, the crash of a replica is transparent to the client process: The client never needs to reissue a request. In primary-backup replication,

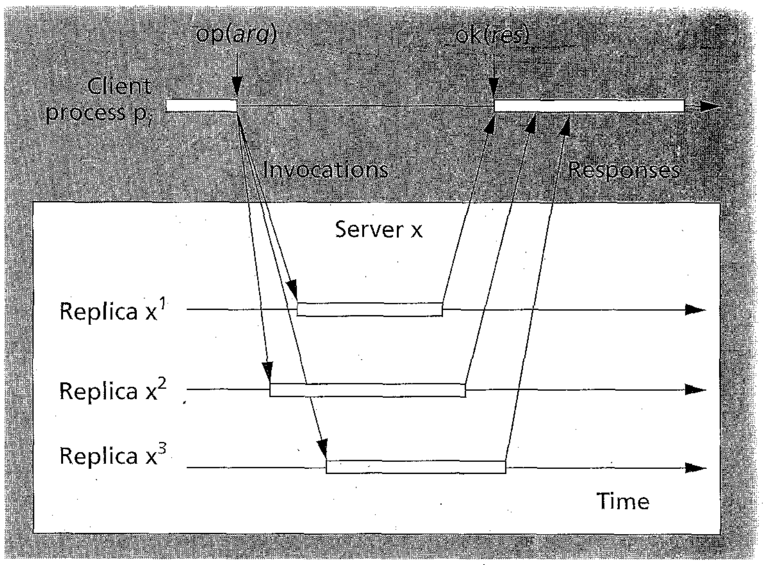

Figure 3. Active-replication technique.

although a backup's crash is transparent to the client, that of the primary replica is not. In that event, the client can experience a significant increase in latency—the time between invocation and the reception of the response.

GROUP COMMUNICATION

The group abstraction, as depicted in Figure 4, is an adequate framework for providing the multicast primitives required to implement both active and primary-backup replication. Several distributed systems provide this abstraction (see the "Group Communication Systems" sidebar). Group g_x, for example, can abstractly represent the set of server x's replicas. The members of g_x are the replicas of x, and g_x can be used to send a message to x's replicas without naming them explicitly.

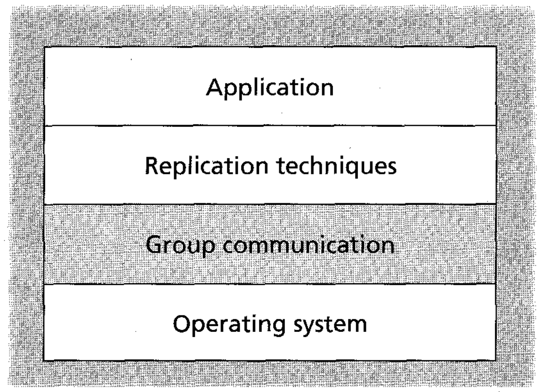

Figure 4. Group communication: the infrastructure for implementing replication.

Static versus dynamic groups

There are two fundamentally different types of groups, static and dynamic. A static group's membership does not change during the system's lifetime. Although we still expect static-group members to crash, a static group does not change its membership to reflect a member's crash. That is, replica x^k remains a member of group g_x after it crashes and before a possible recovery.

A dynamic group's membership changes during the system's lifetime. A group's membership changes, for example, when one of its members crashes. The system removes crashed replica x^k from the group. If x^k recovers later, it rejoins g_x. We use the notion of *view* to model the evolving membership of g_x. The initial membership of g_x is $v_0(g_x)$, and the *i*th membership of g_x is $v_i(g_x)$. Sequence of views $v_0(g_x)$, $v_1(g_x)$, ..., $v_i(g_x)$ thus represents a history of group g_x.[4,5]

Primary-backup replication requires the group's membership to change. If the primary replica crashes. the group must elect a new primary replica. Therefore, this technique uses dynamic groups. In contrast, active replication does not require specific action when a replica crashes, so it can employ static groups. Most existing group communication systems, however, implement active replication using dynamic groups. This is because most of these systems rely on an implementation of total-order multicast that requires dynamic groups. We discuss one exception later.

Group communication and active replication

Active replication requires a total-order multicast primitive. TOCAST (m,g_x) is the total-order multicast of message m to group g_x. Order, atomicity, and termination properties formally define this primitive.

- *Order.* Consider TOCAST (m_1,g_x) and TOCAST (m_2,g_x). If x^j and x^k in g_x deliver m_1 and m_2, they deliver both messages in the same order.
- *Atomicity.* TOCAST(m,g_x) ensures that, if replica x^j in g_x delivers m, then every correct replica of g_x also delivers m.
- *Termination.* Process p_i executes TOCAST(m,g_x). If p_i is correct (does not crash), then every correct replica in g_x eventually delivers m. Termination is a liveness property and ensures system progress.

Group Communication Systems

Isis was the first system to introduce group communication primitives to support reliable distributed applications.[1] Initially developed at Cornell University as an academic project, Isis later became a commercial product, marketed first by Isis Distributed Systems and later by Stratus Computers.

Researchers have developed several other systems.[2] Among these are the toolkits Horus (Cornell University), Transis (Hebrew University, Jerusalem), Totem (University of California, Santa Barbara), Amoeba (Free University, Amsterdam), Consul (University of Arizona, Tucson), Delta-4 (Esprit Project), xAMp (INESC, Lisbon), Rampart (AT&T Bell Labs), Phoenix (Swiss Federal Institute of Technology, Lausanne), Relacs (University of Bologna), Arjuna (University of Newcastle), and Ansaware (APM).

These systems differ in the multicast primitives they provide and in their assumptions about the underlying distributed system. For instance, Delta-4, xAMp, and Arjuna assume a synchronous system with bounded communication delays and process-relative speeds. The others assume an asynchronous system. Transis, Totem, Horus, Phoenix, and Relacs can handle operations in a partitioned network. Rampart considers malicious processes.

References

1. K. Birman, "The Process Group Approach to Reliable Distributed Computing," *Comm. ACM*, Dec. 1993, pp. 37-53.
2. Special Section on Group Communication, D. Powell, ed., *Comm. ACM*, Apr. 1996, pp. 50-97.

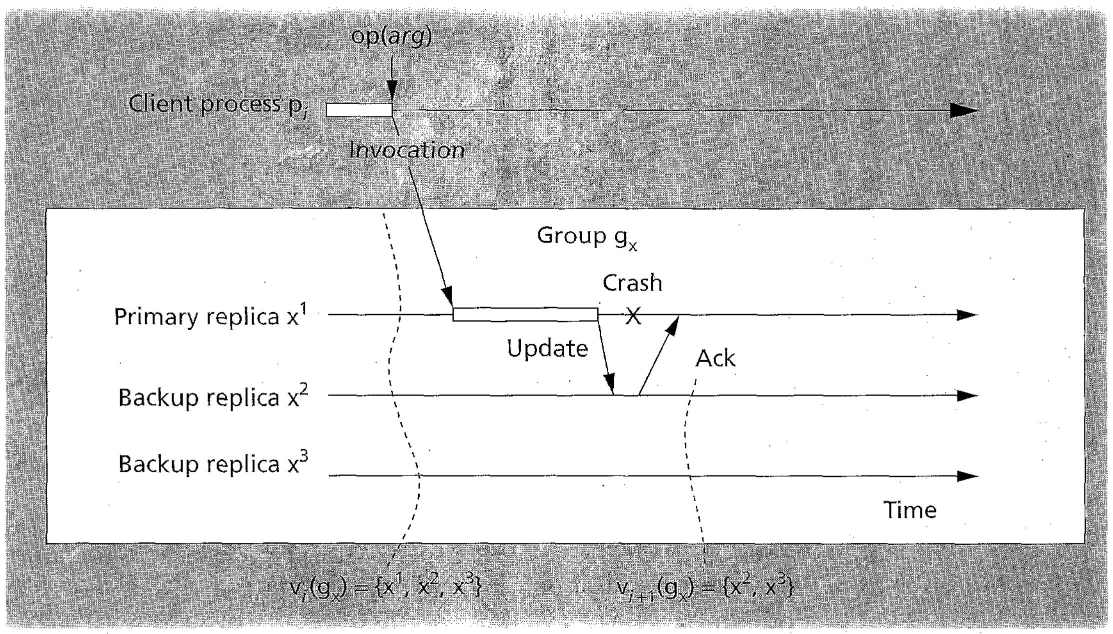

Figure 5. Primary-backup technique; the atomicity problem. Vertical dotted lines represent the time at which replicas deliver new views.

These properties refer to message delivery and not to reception: A process sends a message, which is received by a replica that coordinates with other replicas to guarantee these properties. Finally, the replica delivers the message. The replica performs the operation only after delivery of the message containing the operation. We discuss TOCAST implementations in a later section.

Atomicity and termination properties refer to correct replicas. The notion of correct replicas is a tricky issue in a system model in which replicas can crash and later recover. If replica x^k crashes at some time t, it has no obligation to deliver any message. Later at time $t' > t$, if x^k recovers, it should deliver all messages multicast to g_x before time t'. Yet how can it do so? The state transfer mechanism handles this problem. When replica x^k recovers from a crash, the state transfer mechanism allows x^k to get (from another operational replica x^j in g_x) an up-to-date state, which includes delivery of all messages multicast to g_x. We can use the TOCAST primitive to implement state transfer.

Group communication and primary-backup replication

At first glance, the primary-backup technique appears easier to implement than active replication. Primary-backup replication does not require a TOCAST primitive because the primary replica defines the invocation order. Nevertheless, to correctly handle invocations when the primary replica crashes, the primary-backup technique requires a group communication primitive. This primitive is as difficult to implement as TOCAST. Furthermore, the group must define a new primary replica whenever the current one crashes, so the primary-backup technique requires dynamic groups, which active replication does not.

The sequence of views of group g_x help define the successive primary replicas for server x. For example, for every view, we can define the primary replica as the one with the smallest identification number. Given $v_i(g_x) = \{x^1, x^2, x^3\}$, the primary replica is x^1. In the later view $v_{i+1}(g_x)$, which consists of x^2 and x^3, x^2 becomes the new primary replica. As the system delivers every view to the correct members of g_x, every replica can determine the primary replica's identity.

Having a sequence of views defining a group's history actually makes it irrelevant (from a consistency point of view) whether a replica removed from a view has actually crashed or was incorrectly suspected of doing so. In other words, whether the failure detection mechanism is reliable or not is irrelevant. Unjustified exclusion of a replica decreases the service's fault-tolerance capability, but does not cause inconsistency. If a member of g_x suspects primary replica x^k in view $v_i(g_x)$ of having crashed, and the group defines new view $v_{i+1}(g_x)$, x^k can no longer act as a primary replica. Interaction of x^k with the backup replicas in view $v_{i+1}(g_x)$ reveals the new view (and new primary replica) to x^k.

To summarize, the primary-backup technique uses the primary replica to order invocations, but requires a mechanism that permits group members to agree on a unique sequence of views. This agreement, however, is insufficient to ensure the technique's correctness.

Figure 5 illustrates an example that has an initial view $v_i(g_x) = \{x^1, x^2, x^3\}$; the primary replica is x^1.

- Primary replica x^1 receives an invocation, handles it, and crashes while sending the update message to backups x^2 and x^3. Only x^2 receives the update message.

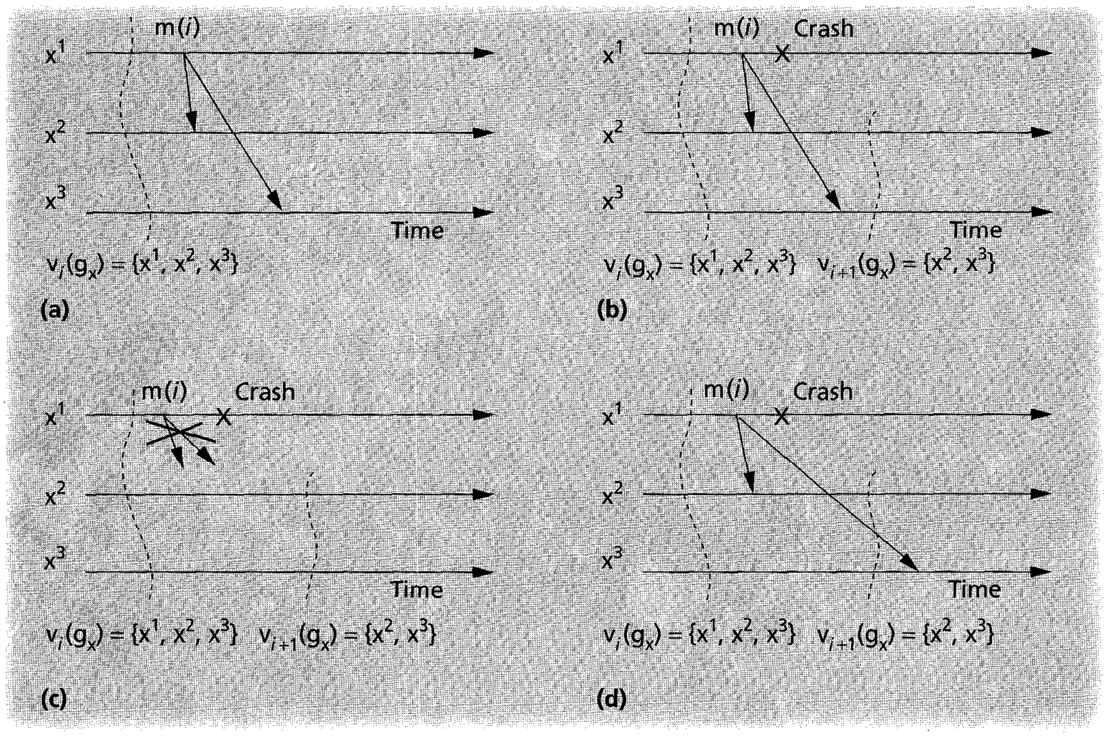

***Figure 6. View-synchronous multicast.** VSCAST allows scenarios 1 (a), 2 (b), and 3 (c), but prevents scenario 4 (d).*

- The group defines new view $v_{i+1}(g_x) = \{x^2,x^3\}$; and x^2 becomes the new primary replica. The states of x^2 and x^3 are inconsistent, however.

The inconsistency stems from the nonatomicity of the update multicast sent by the primary to the backup replicas. Some, but not all, of the backups might receive the update message. We avoid this inconsistency if, whenever the primary replica sends the update message to the backups, either all or none of the correct (noncrashed) backups receive the message. This atomicity semantics, in the context of a dynamic group, is called view-synchronous multicast.[5,6]

View-synchronous multicast (VSCAST)

Dynamic group g_x has sequence of views $v_0(g_x)$, ..., $v_i(g_x)$, $v_{i+1}(g_x)$, and so on. Let $t^k(i)$ be the local time at which replica x^k delivers view $v_i(g_x)$. From $t^k(i)$ on, and until x^k delivers next view $v_{i+1}(g_x)$, x^k time stamps all its messages with current-view number i. Message $m(i)$ is message m with a time stamp for view i.

Let replica x^k multicast message $m(i)$ to all members of view $v_i(g_x)$. View-synchronous multicast (provided by primitive VSCAST) ensures that either all replicas of $v_i(g_x)$ eventually deliver $m(i)$ or that the system defines a new view, $v_{i+1}(g_x)$. In the latter case, VSCAST ensures that either all of the replicas in $v_i(g_x) \cap v_{i+1}(g_x)$ deliver $m(i)$ before delivering $v_{i+1}(g_x)$ or none deliver $m(i)$.

Figure 6 illustrates the definition. In the first scenario, the sender does not crash, and all the replicas deliver $m(i)$. In scenarios 2 and 3, the primary replica crashes, and the system defines a new view. In scenario 2, all the replicas in $v_i(g_x) \cap v_{i+1}(g_x)$ deliver $m(i)$ before delivering new view $v_{i+1}(g_x)$. In scenario 3, no replica in $v_i(g_x) \cap v_{i+1}(g_x)$ delivers $m(i)$. In scenario 4, one replica delivers $m(i)$ in $v_i(g_x)$ and one delivers it in $v_{i+1}(g_x)$. VSCAST semantics prevent scenario 4.

GROUP COMMUNICATION AND CONSENSUS

Several papers have described various implementations of the TOCAST and VSCAST primitives. Nevertheless, most of these implementations neglect liveness issues; that is, they do not specify the assumptions under which the given algorithms terminate. This is unsatisfactory in many applications, including safety-critical applications. Liveness is a difficult issue, directly related to the impossibility of solving the consensus problem in asynchronous distributed systems. Implementing TOCAST and VSCAST comes down to solving this problem. For more information, see the "Consensus in Asynchronous Systems with Unreliable Failure Detectors" sidebar, next page.

With failure detectors, we can use the solution to the consensus problem to implement TOCAST and VSCAST primitives. This leads to implementations that clearly specify the minimal conditions that guarantee termination.

One algorithm implements TOCAST based on consensus.[7] This algorithm launches multiple, independent instances of the consensus problem, identified by integer k. Each of these k consensus instances decides on a batch of messages, batch(k). The processes deliver batch(k) messages before batch(k+1) messages. They also deliver the messages of batch(k) in some deterministic order (the order defined by their identifiers, for example).

The transformation from view-synchronous multicast (VSCAST) to consensus is more complicated than the transformation from TOCAST to consensus. The solution also consists of launching multiple, indepen-

Consensus in Asynchronous Systems with Unreliable Failure Detectors

We define the consensus problem over set of processes P. Every process p_i in P initially proposes a value v_i taken from a set of possible values. The processes in P have to decide on a common value v, such that the following properties hold[1]:

- *Agreement.* No two correct processes decide on different values.
- *Validity.* If a process decides on v, then some process proposed v.
- *Termination.* Each correct process eventually makes a decision.

In 1985, Michael Fischer, Nancy Lynch, and Michael Paterson showed that no deterministic algorithm solves consensus in an asynchronous system in which even a single process can crash.[2] Known as the FLP impossibility result, it also applies to the total-order and view-synchronous multicast problems. Adding time-outs to the asynchronous model to detect crashed processes is insufficient to overcome the impossibility result, and additional assumptions about suspicions are needed. Tushar Chandra and Sam Toueg stated these assumptions in their 1991 work on unreliable failure detectors.[1] Their model attaches a failure detector module to every process in the system. Each of these modules maintains a list of processes currently suspected to have crashed.

Chandra and Toueg defined various classes of failure detectors. Among these, the class of eventually strong failure detectors, denoted $\Diamond S$, is of particular interest. The properties of this class are *strong completeness* and *eventual weak accuracy*.

The $\Diamond S$ failure detector class allows us to solve consensus in an asynchronous system with a majority of correct processes.[1] In situations in which no process is suspected and no process crashes (the most frequent case in practice), solving consensus requires three communication steps (phases). We can even optimize the algorithm to require only two communication steps.[3]

References

1. T. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, Mar. 1996, pp. 225-267.
2. M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *J. ACM*, Apr. 1985, pp. 374-382.
3. A. Schiper, "Early Consensus in an Asynchronous System with a Weak Failure Detector," *Distributed Computing*, Mar./Apr. 1997.

dent instances of consensus. However, consensus k decides not only on a batch(k), but also on the next view's membership. Each process, after learning the decision of consensus k, delivers the remaining, undelivered messages of batch(k) and then delivers the next view. We discuss the details in an earlier work.[8]

The relationships we have discussed between, for example, primary-backup replication and view-synchronous multicast, illustrate the convergence of replication techniques and group communications. These relationships clarify some important issues in fault-tolerant distributed systems. Combined with the link between group communications and the consensus problem, they will certainly lead to interesting developments and new modular implementations. ❖

References

1. M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems,* July 1990, pp. 463-492.
2. N. Budhiraja et al., "The Primary-Backup Approach," in *Distributed Systems,* S. Mullender, ed., ACM Press, New York, 1993, pp. 199-216.
3. F.B. Schneider, "Replication Management Using the State-Machine Approach," in *Distributed Systems,* S. Mullender, ed., ACM Press, New York, 1993, pp. 169-197.
4. A.M. Ricciardi and K.P. Birman, "Using Process Groups to Implement Failure Detection in Asynchronous Environments," *Proc. 10th ACM Symp. Principles Distributed Computing,* ACM Press, New York, 1991, pp. 341-352.
5. K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," *ACM Trans. Computer Systems,* Aug. 1991, pp. 272-314.
6. A. Schiper and A. Sandoz, "Uniform Reliable Multicast in a Virtually Synchronous Environment," *Proc. IEEE 13th Int'l Conf. Distributed Computing Systems,* IEEE CS Press, Los Alamitos, Calif., 1993, pp. 561-568.
7. T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM,* Mar. 1996, pp. 225-267.
8. R. Guerraoui and A. Schiper, "Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems," *Proc. IEEE 26th Int'l Symp. Fault-Tolerant Computing,* IEEE CS Press, Los Alamitos, Calif., 1996, pp. 168-177.

***Rachid Guerraoui** is a lecturer and research associate at the Federal Institute of Technology in Lausanne (EPFL). His current research interests are fault-tolerant distributed systems, transactional systems, and object-oriented computing. Guerraoui received a PhD in computer science from the University of Orsay, France.*

***André Schiper** is a professor of computer science at the EPFL and leads the Operating Systems Laboratory. His current research interests are in fault-tolerant distributed systems and group communication, which have led to development of the Phoenix group communication middleware. He is a member of the ESPRIT Basic Research Network of Excellence in Distributed Computing Systems Architectures (CaberNet).*

Contact Guerraoui or Schiper at Département d'Informatique, Ecole Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland; guerraoui@di.epfl.ch; schiper@di.epfl.ch.