Fault-Tolerance by Replication in Distributed Systems

Rachid Guerraoui André Schiper

Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
1015 Lausanne, Switzerland

Abstract. The paper is a tutorial on fault-tolerance by replication in distributed systems. We start by defining *linearizability* as the correctness criterion for replicated services (or objects), and present the two main classes of replication techniques: *primary-backup replication* and *active replication*. We introduce *group communication* as the infrastructure providing the adequate multicast primitives to implement either primary-backup replication, or active replication. Finally, we discuss the implementation of the two most fundamental group multicast primitives: total order multicast and view synchronous multicast.

1 Introduction

Dependability, i.e. reliability and availability, is one of the biggest trends in software technologies. In the past, it has been considered acceptable for services to be unavailable because of failures. This is rapidly changing: the requirement for high software reliability and availability is continually increasing in domains such as finance, booking-reservation, industrial control, telecommunication, etc. One solution for achieving fault-tolerance is to build software on top of fault-tolerant (replicated) hardware. This may indeed be a viable solution for some application classes, and has been successfully pursued by companies such as Tandem and Stratus. Economic factors have, however, motivated the search for cheaper software-based fault-tolerance, i.e. software-based replication. While this principle is readily understood, the techniques required to implement replication pose difficult problems. The paper presents a survey of the techniques that have been developed, since the mid-eighties, to implement replicated services (called also objects).

To discuss fault-tolerance, one need to specify the types of failures that are considered. Let us assume the following general model of a distributed system: the system consists of a set of *processes* connected through communication *links*, which are used by the processes to exchange *messages*. A process can either be *correct*, or *incorrect*. A correct process is a process that behaves according to its specification. An incorrect process is either a process that crashes, or a process that behaves maliciously. A crashed process stops receiving or sending messages, while a malicious process might send messages that do not follow the specification. We consider in the paper only crash failures, and ignore malicious behavior (also called *Byzantine* failures).

When considering the behavior of communication links, we distinguish two basic system models: the *asynchronous* system model, and the *synchronous* system model. The synchronous system model assumes that the transmission delay of the messages sent over the communication links is bounded by a known value. The asynchronous system model does not set any bound on the transmission delay of messages. This makes the asynchronous model very attractive from a practical point of view, and we consider this model in the paper. The only property that we require from the communication links is *channel reliability*: a message m sent by a process p_i to a process p_j is eventually received by p_j, if both p_i and p_j are correct. Thanks to the possibility of message retransmission, the channel reliability property does not prevent link failures, if we assume that link failures are eventually repaired.

This modelization characterizes the distributed system that we consider. The rest of the paper is organized as follows. Section 2 defines the correctness criterion for replicated servers, called *linearizability*. This criterion gives to the client processes the illusion of non-replicated servers. Section 3 introduces the two main classes of replication techniques that ensure linearizability. These two classes are the *primary-backup* technique, and the *active replication* technique. Section 4 introduces group communication as the framework for the definition of the multicasts primitive required to implement primary-backup replication and active replication. Section 5 discusses the implementation of the two most important group multicast primitives: total order multicast and view synchronous multicast. Section 6 concludes the paper by mentioning existing distributed platforms that support replication.

2 Replica consistency

2.1 Basic model and notations

We consider a set of sequential processes $P = \{p_1, p_2, \ldots, p_n\}$ interacting through a set X of objects. An object has a state accessed by the processes through a set of operations. An operation by a process p_i on an object $x \in X$ is a pair invocation/response. After issuing an invocation, a process is blocked until it receives the matching response. The operation invocation is noted $[x\ op(arg)\ p_i]$, where arg are the arguments of the operation op. The operation response is noted $[x\ ok(res)\ p_i]$, where res is the result returned. The pair invocation/response is noted $[x\ op(arg)/ok(res)\ p_i]$.

In order to tolerate process crash failures, a "logical" object must have several "physical" replicas, located at different sites of the distributed system. The replicas of an object x are noted $x^1, \ldots, x^l$. Invocation of replica x^j located on site s is handled by a process p_j also located on s. We assume that p_j crashes exactly when x_j crashes. Replication is transparent to the client processes, which means that replication does not change the way the invocation of operations, and the corresponding responses, are noted.

2.2 Consistency criteria

A consistency criterion defines the result returned by an operation. It can be seen as a contract between the programmer and the system implementing replication.

Three main consistency criteria have been defined in the literature: linearizability [21], sequential consistency [24] and causal consistency [2]. In all three cases, an operation is performed on the *most recent* state of the object. The three consistency criteria differ however in the definition of the *most recent* state. Linearizability is the most restrictive of the three consistency criteria (linearizability defines the strongest consistency criterion), whereas causal consistency defines the weakest of the three consistency criteria. Both linearizability and sequential consistency define what is informally called a *strong* consistency criterion, whereas causal consistency defines a *weak* consistency criterion. Causal consistency includes sequential consistency (i.e. an execution that satisfies sequential consistency also satisfies causal consistency), and sequential consistency is included in linearizability (i.e. an execution that satisfies linearizability also satisfies sequential consistency).

Most applications require strong consistency, i.e. linearizability or sequential consistency, as it provides the programmers with the illusion of non-replicated objects. We consider here only linearizability. The reason for considering linearizability, rather than sequential consistency, is justified by practical considerations. It turns out that linearizability is easier to implement, rather than just sequential consistency. In other words, most of the implementations of strong consistency turn out to ensure linearizability.

2.3 Linearizability

We give here an informal definition of linearizability. A formal definition can be found in [21]. Let O be an operation, i.e. a pair invocation/response $[x\ op(arg)/ok(res)\ p_i]$. Consider a global real-time clock, and let $t_{inv}(O)$ be the time at which p_i invokes the operation op on object x, and $t_{res}(O)$ the time at which the matching response is received by p_i. Two operations O and O' are said to be sequential, noted $O \prec O'$, if the response of O precedes the invocation of O', i.e. if $t_{res}(O) < t_{inv}(O')$. Two operations O and O' are said to be *concurrent* if neither $O \prec O'$ nor $O' \prec O$ hold. We note $O||O'$ two concurrent operations.

Using the $\prec$ relation, we define linearizability as follows. An execution E is linearizable if there exists a sequence S including all operations of E such that the following two conditions hold:

- for any two operations O and O' such that $O \prec O'$, O appears before O' in the sequence S;
- the sequence S is *legal*, i.e. for every object x, the subsequence of S of which operations are on x, belongs to the sequential specification of x.

To illustrate the definition, consider an object x defining a FIFO queue (initially empty) with the enqueue (noted enq) and dequeue (noted deq) operations, and an execution with the following invocations/responses:

- at time $t = 1$: $[x\ enq(a)\ p_i]$
- at time $t = 2$: $[x\ enq(b)\ p_j]$
- at time $t = 3$: $[x\ ok()\ p_i]$
- at time $t = 4$: $[x\ ok()\ p_j]$
- at time $t = 5$: $[x\ deq()\ p_i]$
- at time $t = 6$: $[x\ deq()\ p_j]$
- at time $t = 7$: $[x\ ok(b)\ p_i]$
- at time $t = 8$: $[x\ ok(a)\ p_j]$

The execution consists thus of four operations: the enqueue operation O_1 by p_i, invoked at global time $t = 1$ and completed at time $t = 3$; the enqueue operation O_2 by p_j, invoked at time $t = 2$ and completed at time $t = 4$; the dequeue operation O_3 by p_i, invoked at time $t = 5$ and completed at time $t = 7$; and the dequeue operation O_4 by p_j, invoked at time $t = 6$ and completed at time $t = 8$. We have $O_1 || O_2$, $O_3 || O_4$, and $O_1, O_2 \prec O_3, O_4$. The above execution is linearizable, as we can exhibit the following legal sequence $S = [O_2, O_1, O_3, O_4]$. The sequence S is legal as it belongs to the sequential specification of a FIFO queue: b is enqueued first (operation O_2), and then a (operation O_1), thus the first dequeue operation O_3 correctly returns b, and the second dequeue operation O_4 correctly returns a.

Consider now that operation O_4 had returned b at time $t = 8$. In this case, the execution is not linearizable as no sequence that belongs to the sequential specification of a FIFO queue, can be constructed from the execution. We discuss in the next section circumstances under which this could have happened.

2.4 Ensuring linearizability

Consider a FIFO queue x, which, in order to be fault-tolerant, is implemented by two replicas x^1, x^2. Consider the execution of Section 2.3, and assume that the replicas observe the following sequence of events:

- replica x^1 receives the invocations in the following order: (1) $[x\ enq(b)\ p_j]$, (2) $[x\ enq(a)\ p_i]$, (3) $[x\ deq()\ p_i]$, (4) $[x\ deq()\ p_j]$;
- replica x^2 receives the invocations in the following order: (1) $[x\ enq(a)\ p_i]$, (2) $[x\ enq(b)\ p_j]$, (3) $[x\ deq()\ p_i]$, (4) $[x\ deq()\ p_j]$.

Each replica handles the invocations sequentially, in the order they are received. Thus replica x^1 sends the responses $ok(b)$ to p_i, and $ok(a)$ to p_j, whereas replica x^2 sends the replies $ok(a)$ to p_i and $ok(b)$ to p_j. If both p_i and p_j consider the responses received from replica x^1, we get the execution of Section 2.3, which is linearizable. However, if p_i considers the response from x^1, whereas p_j considers the response from x^2, then both processes get the response $ok(b)$, and the execution is not linearizable.

The problem with this scenario, is that both replicas x^1 and x^2 do not receive the invocations in the same order. A similar problem can occur if, because of the crash of a client process, one replica, say x^1, handles an invocation, whereas the other replica, i.e. x^2, does not. A sufficient condition to ensure linearizability is to have the replicas agree on the set of invocations they handle, and on the order according to which they handle these invocations These conditions can be expressed more formally as follows:

Atomicity. Given an invocation $[x\ op(arg)\ p_i]$, if one replica of an object x handles this invocation, then every correct (i.e. non-crashed) replica of x also handles the invocation $[x\ op(arg)\ p_i]$.

Order. Given two invocations $[x\ op(arg)\ p_i]$ and $[x\ op(arg)\ p_j]$, if two replicas x^1 and x^2 handle both invocations, they handle them in the same order.

3 Replication techniques

We have introduced linearizability as the correctness criterion for replicated objects. We present in this section two fundamental classes of techniques that ensure linearizability: (1) the *primary-backup* replication technique, and (2) the *active replication* technique. In the first technique, one process, called the *primary*, ensures a centralized control. There is no such centralized control in the second technique. We then present *read/write* techniques that have been designed in the context of file systems and databases.

3.1 Primary-backup replication

In the primary-backup strategy [8], one of the replicas, called the *primary*, plays a special role (Fig. 1): it receives the invocations from the client process, and sends the response back. Given an object x, its primary replica is noted $prim(x)$. The other replicas are called the *backups*. The backups interact with the primary, and do not interact directly with the client process.

Consider the invocation $[x\ op(arg)\ p_i]$ issued by p_i. In the absence of crash of the primary, the invocation is handled as follows:

- Process p_i sends the invocation $op(arg)$ to the replica $prim(x)$.
- The primary $prim(x)$ receives the invocation and performs it. At the end of the operation, the response res is available, and the state of $prim(x)$ is updated. At that point, $prim(x)$ sends the update message ($invId, res, state\text{-}update$) to the backups, where $invId$ identifies the invocation, res is the response, and $state\text{-}update$ describes the state update of the primary, resulting from the invocation $invId$. Upon reception of the update message, the backups update their state, and send back an acknowledgment to the primary (the need for $invId$ and res is discussed below).
- Once the primary has received ack from all correct (i.e. non-crashed) backups, the response is sent to p_i[1].

[1] A primary-backup protocol is called *blocking* if the primary cannot send the reply to the client before having received a certain number of acks from the backups. A non-blocking protocol is possible only under very specific system assumptions [8].

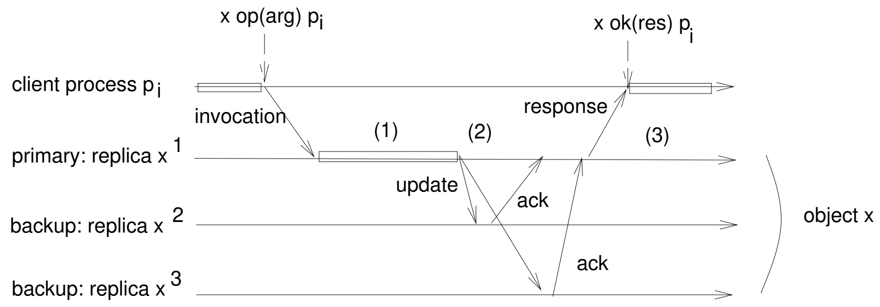

Fig. 1. Primary-backup technique

If the primary does not crash, then the above scheme obviously ensures linearizability: the order in which the primary receives the invocation defines the total order on all the invocations to the object. Ensuring linearizability despite the crash of the primary is more difficult. In the case of the crash of the primary, three cases can be distinguished: (1) the primary crashes before sending the update message to the backups ((1) in Fig. 1), (2) the primary crashes after sending the update message, but before the client receives the response ((2) in Fig. 1), and (3) the primary crashes after the client has received the response ((3) in Fig. 1). In all three cases, a new unique primary has to be selected. In the cases 1 and 2, the client will not receive any response to its invocation, and will suspect a failure. After having learned the identity of the new primary, the client will reissue its invocation. In case 1, the invocation is considered as a new invocation by the new primary. Case 2 is the most difficult case to handle. Atomicity has to be ensured: either all the backups receive the update message, or none of them receive it (we come back to this issue in Section 4). If none of the backups receive the message, case 2 is similar to case 1. If all of the backups receive the update message, then the state of the backups is updated by the operation of the client process p_i, but the client does not get the response, and will reissue its invocation. The information ($invId$, res) is needed in this case, to avoid handling the same invocation twice (that would produce an inconsistent state if the invocation is not idempotent). When the new primary receives the invocation $invId$, rather than handling the invocation, it immediately sends the response res back to the client.

If we assume a perfect failure detection mechanism, apart from the atomicity issue raised above, the primary-backup replication technique is relatively easy to implement. The implementation becomes much more complicated in the case of an asynchronous system model, in which the failure detection mechanism cannot be reliable. The *view-synchronous* communication paradigm, presented in Section 4, defines the communication semantics that ensures correctness of the primary-backup technique in the case of an unreliable failure detection mechanism.

One of the main advantages of the primary-backup technique, is to allow

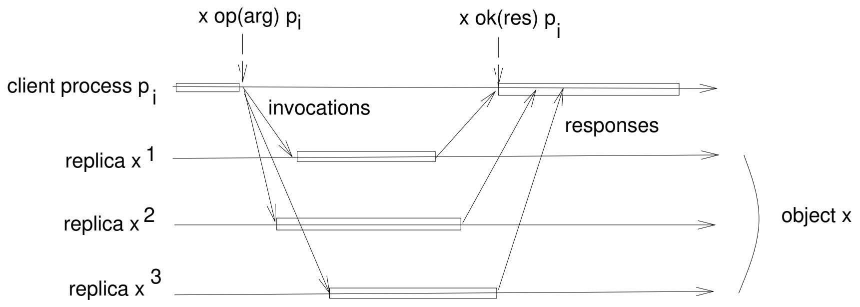

Fig. 2. Active replication technique

for non-deterministic operations. This is not the case with the active replication technique described below.

3.2 Active replication

In the active replication technique, also called "state-machine approach" [33], all replicas play the same role: there is here no centralized control, as in the primary-backup technique. Consider an object x, and the invocation [x $op(arg)$ p_i] issued by p_i (Fig. 2):

- The invocation $op(arg)$ is sent to all the replicas of x.
- Each replica processes the invocation, updates its state, and sends the response back to the client p_i.
- The client waits until either (1) it receives the first response, or (2) it receives a majority of identical responses.
  If the replicas do not behave maliciously (i.e. if Byzantine failures are excluded) then the client process waits only for the first response. If the replicas can behave maliciously (Byzantine failures), then $2f+1$ replicas are needed to tolerate up to f faulty replicas [33]. In this case the client waits to receive $f+1$ identical responses.

The active replication technique requires that the invocations of client processes be received by the non-faulty replicas in the same order. This requires an adequate communication primitive, ensuring the order and the atomicity property presented in Section 2. This primitive is called *total order multicast* or *atomic multicast*. The precise semantics of the total order multicast primitive is given in Section 4.

Apart from the Byzantine failure issue, the tradeoffs between active replication and primary-backup replication are the following:

- Active replication requires the operations on the replicas to be deterministic, which is not the case with the primary-backup technique. "Determinism" means that the outcome of an operation depends only on the initial state

of the replica, and on the sequence of previous operations performed by the replica.
- With active replication, the crash of a replica is transparent to the client process: the client never needs to reissue a request. With the primary-backup technique, the crash of the backups is transparent to the client, but not the crash of the primary. In the case of the crash of the primary, the latency experienced by the client (i.e. the time between the invocation and the reception of the response) can increase significantly. This can be unacceptable for real-time applications.
- The active replication technique uses more resources than the primary-backup technique, as the invocation is processed by every replica.

3.3 Read/write techniques

Several replication techniques have been introduced in the specific context of file systems and databases. These techniques can be viewed as combination of primary backup and active replication techniques, with the additional assumptions that (1) replicated objects can be accessed (only) through *read* and *write* operations, and (2) an underlying concurrency control protocol (ensuring total order) is provided.

The *available copies* replication method [15] ensures atomicity by a *"read one/write all"* technique: a read operation can be performed on any available copy, while a write operation must be executed on all available copies. The *available* copies are defined by a reliable failure detection mechanism. Whenever a copy x^k of some object x crashes, then x^k is removed from the set of available copies. The requirement of a reliable failure detection mechanism clearly means that the technique does not prevent inconsistencies in the case of communication link failures. Quorum methods have been introduced to prevent inconsistencies in the case of link failures. The basic idea was initially introduced by Gifford [14]; it consists in assigning votes to every replica of an object x, and defining read quorums and write quorums such that (1) read quorums and write quorums intersect, and (2) two write quorums intersect. Thus any read operation is performed at least on one replica that has "seen" all the preceding write operations: this ensures the atomicity condition. As we have mentioned above, the ordering condition is assumed to be guaranteed by the underlying transactional system (e.g. through a locking mechanism).

The above technique is called *static voting*, as the read and write quorums do not change during the whole life-time of the system. Static voting has a serious drawback in case of failures, since quorums can be become impossible to obtain. Dynamic voting has been introduced by Davcec and Burkhard [12] to overcome this problem. The basic idea is that after a crash, the system reconfigures to a new subset of replicas, on which new quorums are defined. The dynamic voting techniques have been extended to allow non identical read and write quorums by El Abbadi and Toueg [1]. The quorum technique has been extended to general operations (rather than just read/write operations) by Herlihy [20].

4 Group communication

The group abstraction constitutes the adequate framework for the definition of the multicast primitives required to implement the replication techniques introduced in the previous section. Consider a replicated object x. A group, noted g_x, can abstractly represent the set of replicas of x: the members of g_x are the replicas of x, and g_x can be used to address a message to the set of replicas of x. A group constitutes a convenient logical addressing facility: sending a message to all the replicas of x can be done without explicitly naming the set of replicas of object x.

4.1 Static groups vs dynamic groups

There are two fundamentally different types of groups: *static groups* and *dynamic groups*. A *static* group is a group whose membership does not change during the whole life-time of the system. This does not mean that members of a group g_x are not supposed to crash. It simply means that the membership is not changed to reflect the crash of one of its members: a replica x^k, after its crash, and before a possible recovery, remains a member of the group g_x. Static groups are adequate in the context of active replication, as active replication does not require any specific action to be taken in the case of the crash of one of its replicas. This is not true for the primary-backup replication technique: if the primary crashes, the membership of the group has to be changed, in order to elect a new primary.

A *dynamic* group is a group whose membership changes during the life-time of the system. The membership changes for example as the result of the crash of one of its member: a crashed replica x^k is removed from the group. If x^k later recovers, then it rejoins g_x. The notion of *view* is used to model the evolving membership of g_x. The initial membership of g_x is noted $v_0(g_x)$, and $v_i(g_x)$ is the i^{th} membership of g_x. The history of a group g_x can thus be represented as a sequence of views: $v_0(g_x), v_1(g_x), \ldots, v_i(g_x), \ldots$ [31, 7].

4.2 Group communication and active replication

We have seen in Section 3.2 that active replication requires a total order multicast primitive. Let g_x be a group: we note $TOCAST(m, g_x)$ the total order multicast of message m to the group g_x. This primitive can formally be defined by the following three properties:

Order. Consider the two primitives $TOCAST(m_1, g_x)$ and $TOCAST(m_2, g_x)$, and two replicas x^j and x^k in g_x. If x^j and x^k deliver m_1 and m_2, they deliver both messages in the same order.

Atomicity. Consider the primitive $TOCAST(m, g_x)$. If one replica $x^j \in g_x$ delivers m, then every correct replica of g_x also delivers m.

Termination. Consider the primitive $TOCAST(m, g_x)$ executed by some process p_i. If p_i is correct, i.e. does not crash, then every correct replica in g_x eventually delivers m.

The above properties consider message *delivery* and not message *reception*. Basically, a replica will first receive a message, then perform some coordination with other replicas, to guarantee the above properties, and then *deliver* the message, i.e execute the invoked operation.

The *termination* condition is a *liveness* condition: it prevents the trivial implementation of the *order* and *atomicity* conditions, consisting in never delivering any message. A liveness condition ensures progress of the system. Implementation of the TOCAST primitive is discussed in Section 5.

The above definition of TOCAST uses the notion of a *"correct"* replica. This is a tricky issue in a system model where replicas can crash, and later recover. If a replica x^k has crashed at some time t, then x^k has no obligation to deliver any message. If later, at time $t' > t$, the replica x^k recovers, then x^k should have delivered all messages multicast to g_x up to time t'! This problem is handled by the mechanism called *state transfer*: when a replica x^k recovers after 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, including all the messages that have been TOCAST to g_x.

State transfer. State transfer can be implemented as follows, using the TOCAST primitive. Let x^3 be a replica that recovers after a crash (Fig. 3):

- The replica x^3 starts by executing $TOCAST(\textit{state-req}, g_x)$, where *state-req* is a message requesting the state, and containing the identity of the replica x^3.
- Any replica, upon delivery of *state-req*, sends its current state to x^3. Actually, it is not necessary for every replica to send its state to x^3. In Figure 3, only x^1 sends its state to x^3. We do not discuss such an optimization here. Note that the state is not sent using the TOCAST primitive.
- Replica x^3 waits to deliver its own *state-req* message, ignoring any message delivered before the *state-req* message (e.g. message m_1 in Figure 3). Once the message *state-req* is delivered, x^3 waits to receive the current state from one of the members of g_x. In the meantime, x^3 buffers all the messages delivered after *state-req* (e.g. message m_2 in Figure 3). Upon reception of the "state" message, x^3 initializes its state, and then handles the sequence of buffered messages, (i.e. x^3 updates its state accordingly). Once this is done, replica x^3 handles as usual all the messages delivered after the "state" message.

4.3 Group communication and primary-backup replication

The primary-backup replication scheme does not require a TOCAST primitive. This is because the primary defines the order of the invocations. However, the primary-backup technique requires dynamic groups, in order to define a new primary whenever the current primary has crashed.

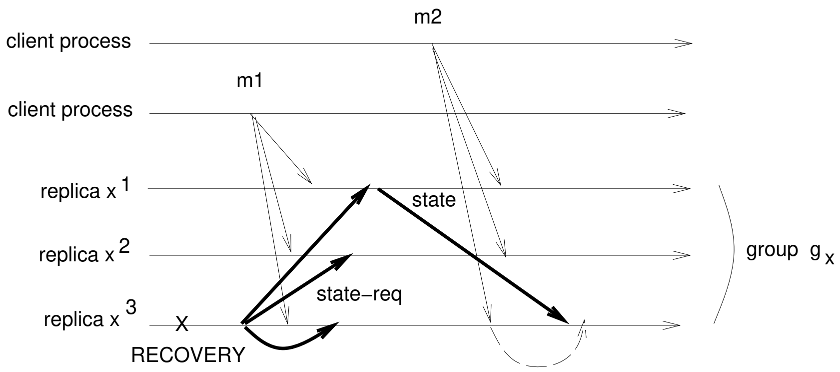

Fig. 3. State transfer (m_1, m_2 and *state-req* are TOCAST messages)

The primary for an object x can easily be defined based on the sequence of views of the group g_x. Assume that in any view $v_i(g_x)$, the replicas are ordered according to some deterministic rule R. The primary can then be defined, for every view, as the first replica according to the rule R. As an example, given $v_i(g_x) = \{x^1, x^2, x^3\}$, and the ordering R defined by the replica's number, the primary for view $v_i(g_x)$ is x^1. If later a new view $v_{i+1}(g_x) = \{x^2, x^3\}$ is defined, replica x^2 becomes the new primary. As every view $v_i(g_x)$ is delivered to all the correct members of g_x, every replica is able to learn the identity of the primary. Notice also that, given the sequence of views defining the history of a group, it is actually irrelevant whether a replica that is removed from a view has really crashed, or was incorrectly suspected to have crashed. In other words, it is irrelevant whether the failure detection mechanism is reliable or not.

To summarize, the primary-backup technique uses the primary to order the invocations, but requires a mechanism to order the views. Ensuring the order on the views is however not sufficient to ensure the correctness of the primary-backup replication technique. To illustrate the problem, consider the following example, with initially the view $v_i(g_x) = \{x^1, x^2, x^3\}$ and the primary x^1 (Fig. 4):

- The primary x^1 receives an invocation, handles it, and crashes while sending the update message to the backups x^2 and x^3. The update message is only received by x^2.
- A new view $v_{i+1}(g_x) = \{x^2, x^3\}$ is defined, and x^2 becomes the new primary. The states of x^2 and x^3 are however inconsistent.

The inconsistency is due to the non-atomicity of the "update" multicast sent by the primary to the backups: the "update" message might be received by some, but not all, of the backups. The inconsistency is avoided if, whenever the primary sends the update message to the backups, either all or none of the correct backups receive the message. This atomicity semantics, in the context of a dynamic membership, is called *view synchronous multicast* [7, 32]. We start by

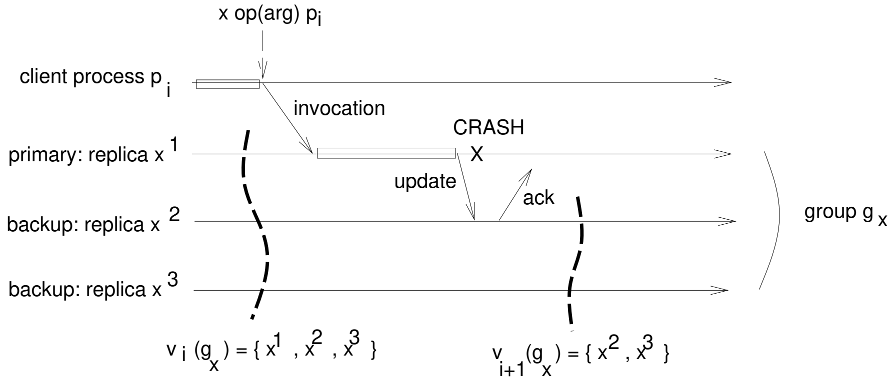

Fig. 4. Primary-backup technique: the atomicity problem (the vertical dotted lines represent the time at which a view is delivered to the replicas)

defining view synchronous multicast, noted VSCAST, and then we show how this semantics ensures consistency of the replicas in the primary-backup technique.

View synchronous multicast. Consider a dynamic group g_x, and a sequence of views $\ldots, v_i(g_x), v_{i+1}(g_x), \ldots$. Let $t^k(i)$ be the local time at which a replica x^k delivers a message containing the composition of the view $v_i(g_x)$. From $t^k(i)$ on, x^k time-stamps all its message with the current view number i. Assume further that every message $m(i)$, time-stamped with the view number i, is multicast to all the members of the view $v_i(g_x)$. Let $v_{i+1}(g_x)$ be the next view. Then either all 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 of them deliver $m(i)$. Figure 5 illustrates the definition. The view synchronous multicast property is satisfied in scenario 1, but neither in scenario 2 nor in scenario 3. In scenario 2, x^2 delivers $m(i)$ whereas x^3 does not. In scenario 3, x^3 delivers $m(i)$, but only after delivering the new view, hence violating the definition of view synchronous multicast (or VSCAST).

To understand that VSCAST actually defines an atomicity condition, define a replica x^k in $v_i(g_x)$ to be "correct" if and only if x^k is also in the next view $v_{i+1}(g_x)$. View atomicity ensures that, given a message $m(i)$ multicast to the members of $v_i(g_x)$, either $m(i)$ is delivered by all the correct members of $v_i(g_x)$, or by none of them. Therefore, if the primary of view $v_i(g_x)$ crashes, and a new view $v_{i+1}(g_x)$ is defined, either all the replicas in $v_{i+1}(g_x)$, or none of them, deliver the last "update" message of the primary. All the replicas in the new view $v_{i+1}(g_x)$ share thus the same state, which ensures consistency.

State transfer. A state transfer mechanism is also required with dynamic groups. In the case of a static group, the state transfer is requested by the recovering replica. With dynamic groups, there is no need for a recovering replica x^k to ask for a state transfer. Instead, upon recovery x^k calls a *join* operation. The join

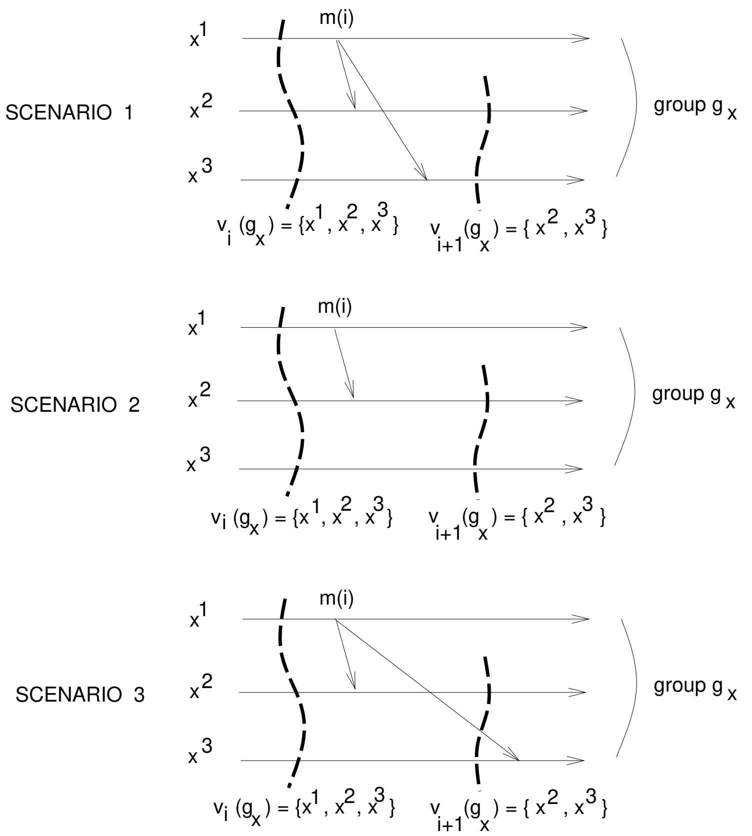

Fig. 5. View synchronous multicast (scenario 1 satisfies the definition whereas scenario 2 and 3 do not)

operation launches the view change protocol, leading to the definition of a new view $v_{i+1}(g_x)$ including x^k. Upon delivery of $v_{i+1}(g_x)$, any member of $v_i(g_x)$, e.g. the primary of view $v_i(g_x)$, sends its state to x^k.

5 Implementation issues

We have given, in Section 4, the specification of the total order multicast primitive TOCAST, required by active replication. We have also defined the view synchronous multicast primitive VSCAST, in the context of dynamic groups and the primary-backup replication technique. We discuss now the implementation of both multicast primitives.

5.1 Total order multicast in asynchronous systems

Many total order multicast algorithms for the asynchronous system model have been proposed in the literature [19]. These algorithms can be classified as being either *symmetric* or *asymmetric*: in a symmetric algorithm all processes perform the same code [6, 25, 10, 4], whereas in an *asymmetric* algorithm one process plays a special role, i.e. defines the ordering of messages [23, 7]. Asymmetric algorithms require less phases and are thus more efficient, but are subject to the contamination problem [16, 35]. Token based algorithms [11, 3] can be classified somewhere in between symmetric and asymmetric algorithms. Moreover, some of these algorithms ([6, 7, 35]) assume the dynamic group model, and an underlying layer implementing view synchronous multicast.

Total order multicast is however related to one fundamental result of fault-tolerant distributed computing: the impossibility of solving the *consensus* problem in asynchronous systems [13] (consensus is defined in Section 5.3). The result, known as the *Fischer-Lynch-Paterson impossibility result* (or *FLP impossibility result*) states that there is no deterministic algorithm that solves consensus in an asynchronous system when *even a single process* can crash. The result applies also to the total order multicast, as both problems are *equivalent* [10]. Equivalence of two problems $\mathcal{A}$ and $\mathcal{B}$ is defined through the concept of reduction [19]: a problem $\mathcal{B}$ reduces to a problem $\mathcal{A}$, if there is an algorithm $\mathcal{T}_{\mathcal{A}\to\mathcal{B}}$ that transforms *any* algorithm for $\mathcal{A}$ into an algorithm for $\mathcal{B}$. Two problems $\mathcal{A}$ and $\mathcal{B}$ are *equivalent* if $\mathcal{A}$ reduces to $\mathcal{B}$ and $\mathcal{B}$ reduces to $\mathcal{A}$. Thus, if two problems are equivalent, whenever one of the two problems can be solved, the other can also be solved.

Because consensus and total order multicast are equivalent, there is no algorithm implementing the TOCAST primitive in an asynchronous system when a single process can crash. This means that, given any algorithm implementing TOCAST, it is always possible to define a run such that one of the three conditions defining TOCAST (Order, Atomicity, Termination) is violated.

We show in Section 5.4 how to get around the FLP impossibility result, by augmenting the asynchronous system model with unreliable *failure detectors*. As we show in Section 5.5, this augmented system model defines also the framework in which the total order multicast problem can be solved, and hence a TOCAST primitive can be implemented.

5.2 View synchronous multicast in asynchronous systems

View synchronous multicast has been introduced by the Isis system [5]: its implementation uses the output of a group membership protocol [31] that delivers the sequence of views of the dynamic group model, and a *flush* protocol [7]. As pointed out in [32], the flush protocol might lead in certain circumstances to violate the view synchronous multicast definition: [32] proposes also a correct implementation of view synchronous multicast.

However it can be shown that consensus reduces to the view synchronous multicast problem: whenever the view synchronous multicast problem can be solved, consensus can also be solved. Hence the FLP impossibility result applies also to the view synchronous multicast problem (and to the implementation of the VSCAST primitive). To circumvent this impossibility result, we have also to consider an asynchronous system model, augmented with unreliable failure detectors. We sketch in Section 5.6 an algorithm based on consensus, that solves the view synchronous multicast problem, and thus implements a VSCAST primitive.

5.3 The consensus problem

The previous sections have pointed out the fundamental role played by the consensus problem in fault-tolerant distributed computing. The consensus problem is defined over a set Π of processes. Every process $p_i \in \Pi$ proposes initially a value v_i taken from a set of possible values (v_i is said to be the initial value of p_i), and the processes in Π have to decide on a common value v such that the following properties hold [10]:

Agreement. No two correct processes decide differently.
Validity. If a process decides v, then v was proposed by some process.
Termination. Each correct process eventually decides.

The agreement condition allows incorrect processes to decide differently from correct processes. A stronger version of the consensus problem, called *uniform consensus*, forbids incorrect processes to decide differently from correct processes. Uniform consensus, is defined by the *uniform agreement* property:

Uniform agreement. No two processes (correct or not) decide differently.

5.4 Failure detectors

In order to overcome the FLP impossibility result, Chandra and Toueg have proposed to augment the asynchronous system model with the notion of (unreliable) failure detector. A failure detector can be seen as a set of (failure detector) modules D_i, one module being attached to every process p_i in the system. Each failure detector module D_i maintains a list of processes that it currently suspects to have crashed. *"Process p_i suspects process p_j"* at some local time t, means that at local time t, process p_j is in the list of suspected processes maintained by D_i. Suspicions are essentially implemented using time-outs, which means that a failure detector module D_i can make mistakes by incorrectly suspecting a process p_j. Suspicions are however not stable. If at a given time D_i suspects p_j, and later learns that the suspicion was incorrect, then D_i removes p_j from its list of suspected processes.

Chandra and Toueg define various classes of failure detectors [10]. Each class is specified by a *completeness* property, and an *accuracy* property. A completeness property puts a condition on the detection of crashed processes, while an accuracy property restricts the mistakes that a failure detector can make. From the failure detector classes defined by Chandra and Toueg, we consider only the class of *eventually strong* failure detectors, noted $\diamond\mathcal{S}$, defined by the following *strong completeness* and *eventual weak accuracy* properties:

Strong completeness. Eventually every crashed process is permanently suspected by every correct process.

Eventual weak accuracy. Eventually some correct process is not suspected by any correct process.

The $\diamond\mathcal{S}$ failure detector class is important, as any failure detector of this class allows to solve consensus in an asynchronous system with a majority of correct processes (i.e. when less then a majority of processes can crash). An algorithm solving consensus under these assumptions is described in [10]. It has been shown that $\diamond\mathcal{S}$ is the weakest class that makes it possible to solve consensus in an asynchronous system with a majority of correct processes [9] [2].

Finally, it has also been shown that any algorithm that solves consensus in an asynchronous system with unreliable failure detectors, also solves the *uniform consensus* problem [17]. Both problems are thus identical under the above assumptions.

5.5 Reduction of total order multicast to consensus

We sketch here the Chandra-Toueg algorithm for total order multicast [10]. The algorithm transforms the total order multicast problem into consensus. Such a transformation is called a *reduction* of total order multicast to consensus. It enables to implement the TOCAST primitive using consensus.

Consider a static group of processes g_x, and messages TOCAST to g_x. The algorithm launches multiple, independent, instances of consensus among the processes in g_x. The various consensus instances are identified by an integer k, and consensus number k decides on a batch of messages noted $batch(k)$. Each process $p_i \in g_x$ delivers the message in the following order:

- the messages of $batch(k)$ are delivered before the messages of $batch(k+1)$;
- for all k, the messages of $batch(k)$ are delivered in some deterministic order (e.g. in the order defined by their identifiers).

The various instances of consensus are defined as follows. Let m be a message TOCAST to g_x. Message m is first multicast to g_x (unordered multicast). When m is received by p_i, it is put into p_i's buffer of undelivered messages, noted $undeliv_i$. Whenever p_i starts a consensus, say consensus number k, p_i's initial value for consensus number k is the current value of $undeliv_i$. Process p_i then executes the consensus algorithm. Once consensus k is solved, i.e. $batch(k)$ is decided, process p_i delivers the messages of $batch(k)$ in some deterministic order, and removes the messages in $batch(k)$ from $undeliv_i$. If at that point $undeliv_i$ is non-empty, p_i starts consensus number $k+1$. Otherwise, p_i starts consensus number $k+1$, only once $undeliv_i$ becomes non-empty.

[2] Actually, the result is proven for the failure detector class $\diamond\mathcal{W}$. However, the failure detector classes $\diamond\mathcal{S}$ and $\diamond\mathcal{W}$ are equivalent [10].

5.6 Reduction of view synchronous multicast to consensus

The transformation from view synchronous multicast to consensus is more complicated than the transformation of total order multicast to consensus. The main ideas are sketched here. Additional details can be found in [18], where the reduction is presented as an instance of the generic paradigm called *Dynamic Terminating Multicast*.

Consider the implementation of view synchronous multicast in a group g_x. The solution consists also in launching multiple, independent, instances of consensus, identified by an integer k. Consensus number k decides however not only on a batch of messages $batch(k)$, but also on the membership for the next view $v_{k+1}(g_x)$. Each process p_i, after learning the decision of consensus number k, first delivers the messages of $batch(k)$ that it has not yet delivered, and then delivers the next view $v_{k+1}(g_x)$. Consensus number k is performed either among the processes from the initial view $v_0(g_x)$, or among the processes of the current view $v_k(g_x)$ [3].

The various instances of consensus are based on the notion of *stable* message. Let m be a message multicast to a view $v_k(g_x)$: $stable_i(m)$, which is a local predicate, is true if and only if p_i knows that every process in $v_k(g_x)$ has received m. Whenever some process $p_i \in v_k(g_x)$ has received a message m, and if after some time-out period $stable_i(m)$ does not hold, then p_i multicasts the message $req\text{-}view(k+1)$ to $view_k(g_x)$, in order to launch the consensus number k that will decide on the next view $v_{k+1}(g_x)$. Every process $p_j \in v_k(g_x)$, when receiving the $req\text{-}view(k+1)$ message, replies by multicasting its non-stable messages: the reply is multicast to the set of processes that solve the consensus problem number k. The way the replies are used to define the initial value for the consensus problem can be found in [18].

6 Concluding remarks

The paper has given a survey of the problems related to achieving fault-tolerance by replication in distributed systems. Linearizability has been introduced as the abstract correctness criterion, and "active replication/primary-backup" have been presented as the two main classes of replication techniques. The total order multicast primitive has then been introduced as the adequate primitive to support active replication, and the view synchronous multicast primitive has been introduced as the adequate primitive to support the primary-backup technique. Finally, the conceptual difficulty of implementing both primitives in an asynchronous system has been related to the Fischer-Lynch-Paterson impossibility result about consensus. As shown by Chandra and Toueg, this impossibility result can be overcome by augmenting the asynchronous system model with unreliable failure detectors that satisfy well defined completeness and accuracy properties. This defines also the framework in which total order multicast and view synchronous multicast primitives can be implemented. The reduction

[3] The two options are actually not equivalent, but the difference is not discussed here.

to consensus constitutes, in this framework, the right way to implement these primitives.

The real issue in achieving fault-tolerance by replication is thus related to the implementation of the group multicast primitives. This has led to the development of "group communication platforms", which provide the application programmer with the adequate multicast primitives required to implement replication. Isis is the best known among such systems [5]. Initially developed at Cornell University as an academic project, Isis has later become a commercial product, marketed first by Isis Distributed Systems (IDS) and subsequently by Stratus Computers. Other platforms that have been built around the world include: Horus (Cornell University) [34], Transis (Hebrew University, Jerusalem) [26], Totem (University of California, Santa Barbara) [29], Amoeba (Free University, Amsterdam) [22], Consul (University of Arizona, Tucson) [28], Delta-4 (Esprit Project) [30], Phoenix (Federal Institute of Technology, Lausanne) [27]. All these systems, except Delta-4, assume an asynchronous system model.

Despite the existence of these various platforms, none of them provide the final answer that the application needs. Some of these systems are architectured in an unnecessarily complex way, some are difficult to use, some do not offer the right abstractions to the application programmer, some do not ensure correctness in some specific scenario cases, and finally most of them do not provide precise characterization of the conditions under which liveness is ensured in the system. The design and implementation of adequate group communication platforms remains an interesting active research area.

References

1. A. El Abbadi and S. Toueg. Maintaining Availability in Partitioned Replicated Databases. *ACM Trans. on Database Systems*, 14(2):264–290, June 1989.
2. M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. TR GIT-CC-93/55, Georgia Institute of Technology, July 94.
3. Y. Amir, L.E. Moser, P.M. Melliar-Smith, D.A. Agarwal, and P.Ciarfella. Fast Message Ordering and Membership Using a Logical Token-Passing Ring. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 551–560, May 1993.
4. E. Auceaume. *Algorithmique de Fiabilisation de Systèmes Répartis.* PhD thesis, Université de Paris-Sud, Centre d'Orsay, January 1993.
5. K. Birman. The Process Group Approach to Reliable Distributed Computing. *Comm. ACM*, 36(12):37–53, December 1993.
6. K. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. on Computer Systems*, 5(1):47–76, February 1987.
7. K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. on Computer Systems*, 9(3):272–314, August 1991.
8. N. Budhiraja, K. Marzullo, F.B. Schneider, and S. Toueg. The Primary-Backup Approach. In Sape Mullender, editor, *Distributed Systems*, pages 199–216. ACM Press, 1993.
9. T.D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. Technical report, Department of Computer Science, Cornell

University, May 1994. A preliminary version appeared in the *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 147–158. ACM Press, August 1992.
10. T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 95-1535, Department of Computer Science, Cornell University, August 1995. A preliminary version appeared in the *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.
11. J. M. Chang and N. Maxemchuck. Reliable Broadcast Protocols. *ACM Trans. on Computer Systems*, 2(3):251–273, August 1984.
12. D. Davcec and A. Burkhard. Consistency and Recovery Control for Replicated Files. In *Proceedings of the 10th Symposium on Operating Systems Principles*, pages 87–96, 1985.
13. M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of ACM*, 32:374–382, April 1985.
14. D.K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–159, December 1979.
15. N. Goodmand, D. Skeen, A. Chan, U. Dayal, S. Fox, and D. Ries. A recovery algorithm for a distributed database system. In *Proc. of the 2nd ACM SIGATC-SIGMOD Symposium on Principles of Database Systems*, March 1983.
16. A. S. Gopal. *Fault-Tolerant Broadcast and Multicasts: The Problem of Inconsistency and Contamination.* PhD thesis, Cornell University, Ithaca, NY, March 1992.
17. R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In *9th Intl. Workshop on Distributed Algorithms (WDAG-9)*, pages 87–100. Springer Verlag, LNCS 972, September 1995.
18. R. Guerraoui and A. Schiper. Transaction model vs Virtual Synchrony Model: bridging the gap. In *Theory and Practice in Distributed Systems*, pages 121–132. Springer Verlag, LNCS 938, 1995.
19. V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In Sape Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press, 1993.
20. M. Herlihy. A Quorum-Consensus Replication Method for Abstract Data Types. *ACM Trans. on Computer Systems*, 4(1):32–53, February 1986.
21. M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. on Progr. Languages and Syst.*, 12(3):463–492, 1990.
22. M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *IEEE 11th Intl. Conf. Distributed Computing Systems*, pages 222–230, May 1991.
23. M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and H. E. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.
24. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C28(9):690–691, 1979.
25. S. W. Luan and V. D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. *IEEE Trans. Parallel & Distributed Syst.*, 1(3):271–285, July 90.
26. D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis approach to high available cluster communication. Technical Report CS-94-14, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.
27. C. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks.* PhD thesis, Federal Institute of Technology, Lausanne (EPFL), 1996. To appear.

28. S. Mishra, L.L. Peterson, and R. D. Schlichting. Consul: a communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1:87–103, 1993.
29. L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended Virtual Synchrony. In *IEEE 14th Intl. Conf. Distributed Computing Systems*, pages 56–67, June 1994.
30. D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.
31. A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Proc. of the 10th ACM Symposium on Principles of Distributed Computing*, pages 341–352, August 1991.
32. A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *IEEE 13th Intl. Conf. Distributed Computing Systems*, pages 561–568, May 1993.
33. F.B. Schneider. Replication Management using the State-Machine Approach. In Sape Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.
34. R. van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. The Horus System. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing with the Isis Toolkit*, pages 133–147. IEEE Computer Society Press, 1993.
35. U. Wilhelm and A. Schiper. A Hierarchy of Totally Ordered Multicasts. In *14th IEEE Symp. on Reliable Distributed Systems (SRDS-14)*, pages 106–115, Bad Neuenahr, Germany, September 1995.