Chapter 11

Availability

For a system that provides a service, *availability* is defined as the ability of the system to actually deliver the service. While several causes may hamper this ability, this chapter concentrates on the methods of improving system availability in the face of failures. *Fault tolerance* covers the concepts and tools that aim at achieving this goal.

The chapter starts with a review of the main concepts and techniques related to fault tolerance. It specifically examines the case of distributed systems, with special emphasis on group communication, which plays a central role in this area. It goes on with the application of these techniques to two main problems: improving the availability of servers, and improving the availability of data. It then examines the notions related to partial availability, and concludes with two case studies.

11.1 Main Concepts and Terminology

A system is *available* if it is ready to correctly deliver the service that it is designed to provide. Availability may be hindered by several causes, among which:

- failures, i.e., accidental events or conditions that prevent the system, or any of its components, from meeting its specification;
- overload, i.e., excessive demand on the resources of the system, leading to service degradation or disruption;
- security attacks, i.e., deliberate attempts at disturbing the correct operation of the system, causing malfunction or denial of service.

In this chapter, we examine how availability can be maintained in the presence of failures. Overload conditions are discussed in Chapter 12 and security attacks in Chapter 13.

This section presents a few elementary notions related to fault tolerance: availability and reliability measures (11.1.1); basic terminology (11.1.2); models of fault tolerance (11.1.3); availability issues for middleware (11.1.4). Section 11.2 examines the main aspects of fault tolerance: error detection, error recovery, and fault masking. This presentation is only intended to provide the minimal context necessary for this chapter, not to cover the subject in any depth. For a detailed study, see Chapter 3 of [Gray and Reuter 1993].

11.1.1 Measuring Availability

The measure of availability applies to a system as a whole or to any of its individual components (we use the general term "system" in the following). A system's availability depends on two factors:

- The intrinsic failure rate of the system, which defines the system's *reliability*.
- The mean time needed to repair the system after a failure.

The reliability of a system is measured by the mean time that the system runs continuously without failure. More precisely, consider a system being started (or restarted) at some instant, and let f be the length of time till its first failure. This is a random variable (since failures are mostly due to random causes). The mean (or expected value) of this variable f is called the Mean Time To Failure ($MTTF$) of the system.

Depending on the causes of the failures, the $MTTF$ of a system may vary with time (e.g., if failures are related to aging, the $MTTF$ decreases with time, etc.). A current assumption (not valid for all systems) is that of failures caused by a memoryless process: the probability rate of failure per unit time is a constant, independent of the past history of the system. In that case, the $MTTF$ is also a constant[1].

Likewise, the repair rate is measured by the expected value of the time needed to restore the system's ability to deliver correct service, after a failure has occurred. This is called the Mean Time To Repair ($MTTR$). One also defines Mean Time Between Failures ($MTBF$) as $MTTF + MTTR$. In the rest of this section, we assume that both the $MTTF$ and the $MTTR$ are constant over time.

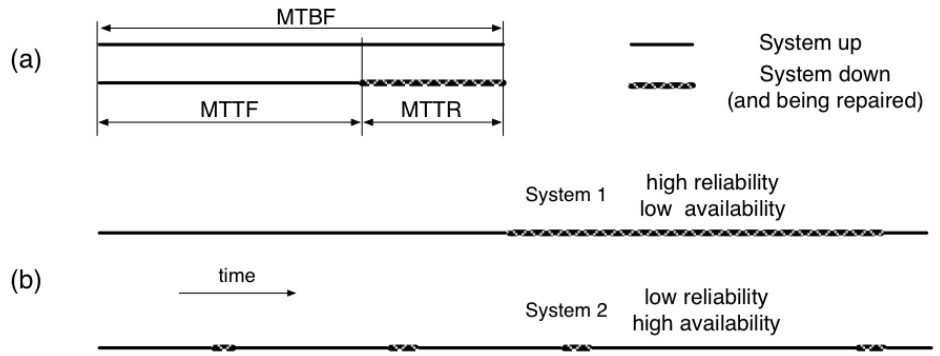

Figure 11.1. Reliability vs Availability

Assume a failure mode (the fail-stop mode, see 11.1.3) in which the system is either correctly functioning or stopped (and being repaired). Define the availability a of a system

[1]For a memoryless failure process, let p be the probability of failure per unit time. Then, if $p \ll 1$, $MTTF \approx 1/p$.

as the mean fraction of time that the system is ready to provide correct service. Then (see Figure 11.1a), the relationship between availability and reliability is given by:

$$a = MTTF/(MTTF + MTTR) = MTTF/MTBF.$$

The distinction between reliability and availability is further illustrated in Figure 11.1b.

To emphasize the importance of the repair time, note that the *un*availability rate of a system is $1-a = MTTR/(MTTF+MTTR) \approx MTTR/MTTF$, because in most practical situations $MTTR \ll MTTF$. Thus reducing $MTTR$ (aiming at quick recovery) is as important as increasing $MTTF$ (aiming at high reliability). This remark has motivated, among others, the Recovery Oriented Computing initiative [ROC 2005].

In practice, since the value of a for a highly available system is a number close to 1, a common measure of availability is the "number of 9s" (e.g., five nines means $a = 0.99999$, a mean down time of about 5 minutes per year).

11.1.2 Failures, Errors and Faults

In order to clarify the concepts of dependability (a notion that covers several qualities, including fault tolerance and security), and to define a standard terminology for this area, a joint committee of the IEEE and the IFIP was set up in 1980 and presented its proposals in 1982. A synthetic presentation of the results of this effort is given in [Laprie 1985]. A more recent and more complete document is [Avižienis et al. 2004]. These definitions are now generally accepted.

We briefly present the main notions and terms related to fault tolerance (a summary is given in 10.1.3).

A system is subject to a *failure* whenever it deviates from its expected behavior. A failure is the consequence of the system being in an incorrect state; any part of a system's state that fails to meet its specification is defined as an *error*. An error, in turn, may be due to a variety of causes: from human mistake to malfunction of a hardware element to catastrophic event such as flood or fire. Any cause that may lead to an error, and therefore may ultimately occasion a failure, is called a *fault*.

The relationship between fault and error and between error and failure is only a potential one. A fault may exist without causing an error, and an error may be present without causing a failure. For example, a programming error (a fault) may lead to the corruption of an internal table of the system (an error); however the table being in an incorrect state does not necessarily lead to a failure. This is because the incorrect values may never be used, or because the internal redundancy of the system may be such that the error has no consequence on the system's behavior. A fault that does not cause an error is said to be *dormant*; likewise, an undetected error is said to be *latent*. If the failure does occur, the time delay between the occurrence of the fault and the failure is called the latency. A long latency makes it difficult to locate the origin of a failure.

In a complex system, a fault which affects a component may cause the system to fail, through a propagation mechanism that may be summarized as follows (Figure 11.2).

Suppose component A depends on component B, i.e., the correct operation of A relies on the correct provision of a service by B to A. A fault that affects B may cause an error in B, leading to the failure of B. For A, the inability of B to deliver its service is a fault.

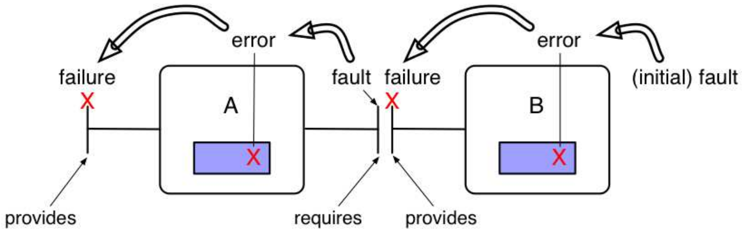

Figure 11.2. Error propagation

This fault may lead to an error in A, possibly causing the failure of A. If A is itself used by another component, the error may propagate further, and eventually lead to the failure of the whole system.

How to prevent the occurrence of failures? The first idea is to try to eliminate all potential causes of faults. However, due to the nature and variety of possible faults (e.g., human mistakes, hardware malfunction, natural causes, etc.), perfect fault prevention is impossible to achieve. Therefore, *systems must be designed to operate in the presence of faults*. The objective is to prevent a fault from provoking a failure, and, if a failure does occur, to recover from it as fast as possible. Methods for achieving this goal are presented in 11.2.

11.1.3 Models for Fault Tolerance

Recall (see 1.4.2) that a model is a simplified representation of (part of) the real world, used to better understand the object being represented. Models are specially useful in the area of fault tolerance, for the following reasons:

- It is convenient to have a framework to help classify and categorize the faults and failures, because of their wide variety.
- The tractability of several problems in fault tolerance strongly depends on the hypotheses on the structure and operation of the system under study. A model helps to accurately formulate these assumptions.

We represent the system as a set of components which communicate through messages over a communication system (see Chapter 4).

Here is a (simplified) classification of failures, in increasing degree of severity (see e.g., [Avižienis et al. 2004] for more details, and for a taxonomy of faults). The component affected by the failure is referred to as “the component”.

- The simplest failure mode is called *fail-stop*. When such a failure occurs, the component immediately stops working (and therefore stops receiving and sending messages). Thus, in the fail-stop mode, a component is either operating correctly or inactive[2].

[2] Since fail-stop is a simple, well understood behavior, a common technique is to force any failure to this mode, by stopping a faulty component as soon as an error is detected, and by signaling the failure to the users of the component. This technique is called *fail-fast*.

- In *omission* failures, the component may fail to send and/or to receive some messages. Otherwise its behavior is normal. This failure mode may be useful to simulate a malfunction of the communication system.
- *Timing* failures only affect the temporal behavior of the component (e.g., the time needed to react to a message).
- In the most general failure mode, the behavior of the failed component is totally unrestricted. For historical reasons [Lamport et al. 1982], this kind of failures is called *Byzantine*. Considering Byzantine failures is useful for two reasons. From a theoretical point of view, this failure mode is the most difficult to handle. From a practical point of view, solutions devised for Byzantine failures may be used in the most extreme conditions, e.g., for critical applications in a hostile environment, or for systems subject to attacks.

The behavior of the communication system has a strong influence on fault tolerance techniques. We consider two main issues: asynchrony and message loss.

Regarding asynchrony, recall (4.1.2) that a distributed system, consisting of nodes linked by a communication system, may be synchronous or asynchronous. The system is *synchronous* if known upper bounds exist for the transmission time of an elementary message, and for the relative speed ratio of any two nodes. If no such bounds exist, the system is *asynchronous*. Intermediate situations (*partially synchronous* systems) have also been identified [Dwork et al. 1988]. Some form of synchrony is essential in a distributed system, because it allows the use of timeouts to detect the failure of a remote node, assuming the communication link is reliable (details in 11.4.1). In an asynchronous system, it is impossible to distinguish a slow processor from a faulty one, which leads to impossibility results for some distributed algorithms in the presence of failures (details in 11.3.3).

Regarding message loss, the main result [Akkoyunlu et al. 1975], often known as the Generals' paradox [Gray 1978], is the following. Consider the coordination problem between two partners, A and B, who seek agreement on a course of action (both partners should perform the action, or neither). If the communication system linking A and B is subject to undetected message loss, no terminating agreement protocol can be designed. The communication protocols used in practice (such as TCP) do not suffer from this defect, because they implicitly assume a form of synchrony. Thus the loss of a message is detected by acknowledgment and timeout, and the message is resent till either it is successfully received or the link is declared broken.

11.1.4 Working Assumptions

We are specifically interested in availability support at the middleware level. Middleware sits above the operating system and networking infrastructure, and provides services to applications. Thus we need to examine (a) the assumptions on the properties of the infrastructure; and (b) the availability requirements placed by applications on the middleware layer.

Middleware most frequently uses the communication services provided by the network transport level (4.3.2). The guarantees ensured by these services vary according to the nature of the network (e.g., a cluster interconnect, a LAN, the global Internet, a wireless network). However, for all practical purposes, we make the following assumptions in the rest of this chapter:

- The communication system is *reliable*, i.e., any message is eventually delivered uncorrupted to its destination, and the system does not duplicate messages or generate spurious messages.
- The middleware system is *partially synchronous*, in the following sense: there exists bounds on the transmission time of a message and on the relative speed of any two processes, but these bounds are unknown a priori and only hold after some time.

  This assumption is needed to escape the impossibility results (see 11.3.3) that hold in asynchronous systems; its validity is discussed in 11.4.1.

Different assumptions may be needed in specific cases, e.g., message loss and network partition in the case of wireless networks.

We assume a fail-stop (or fail-fast) failure mode for components, be they hardware or software. Again, different assumptions may hold in specific situations. Since a system is made of a (potentially large) number of components, a common situation is one in which some part of the system is subject to failure. The goal is then to preserve the ability of the system to deliver its service, possibly with a degraded quality. This issue is developed in 11.8.

According to the "end to end principle" ([Saltzer et al. 1984], see also 4.2.2), some functions, including those related to fault tolerance, are best implemented at the application level, since the most accurate information about the application state and the goals to be pursued is available at that level. The role of middleware with respect to fault tolerance is to provide the tools to be used by the applications to achieve specified availability guarantees. To do so, the middleware layer uses the communication primitives of the transport level, such as specified above. For example (see 11.4), the middleware layer uses reliable point to point transmission to implement reliable broadcast and atomic broadcast, which are in turn used to improve availability at the application level.

What are the main causes of failures in actual systems? Two landmark papers are [Gray 1986], which analyzes failures in the Tandem fault-tolerant servers, and [Oppenheimer et al. 2003], which investigates failures in large scale Internet services. Both studies observe that system administration errors are the dominant cause of failure (about 40% for servers, 35% for Internet services). Most of these administration errors are related to system configuration. Software failures account for about 25% in both studies. The part of hardware is about 18% for servers, 5 to 10% for Internet services. The proportion of network failures in Internet services depends on the nature of the application, in the range of 15 to 20%, and may reach 60% for "read mostly" services. Contrary to the fail-stop assumption, network failures tend to be gradual, and are less easily masked than hardware or software failures.

11.2 Aspects of Fault Tolerance

Fault tolerance is based on a single principle, *redundancy*, which may take various forms:

- Information redundancy. Adding redundant information helps detecting, or even correcting, errors in storage or in transmission. In particular, our assumption on reliable transmission is based on the use of error detecting or correcting codes, together with sending retries, at the lower levels of the communication protocols.

- Temporal redundancy. This covers the use of replicated processing: the same action is performed in several instances, usually in parallel, to increase the probability that it will be achieved at least once in spite of failures. The application of this principle is developed in 11.6.

- Spatial redundancy. This covers the use of replicated data: information is maintained in several copies to reduce the probability of data loss or corruption in the presence of failures. The application of this principle is developed in 11.7.

There are many ways to exploit these forms of redundancy, leading to several fault tolerance techniques, which in turn can be combined to ensure system availability. The main basic techniques can be categorized as follows.

Error detection is the first step of any corrective course related to fault tolerance. It identifies the presence of an error (a system state that does not satisfy its specification), and triggers the further operations. Error detection is further detailed in 11.2.1.

Error compensation (or fault masking) consists in providing the system with enough internal redundancy so as to cancel the effects of an error, thus preventing the occurrence of a fault from causing the system to fail. Masking is further detailed in 11.2.3.

If failure actually occurs, *error recovery* aims at restoring the system's ability to deliver correct service by eliminating the errors that caused the failure. Two main approaches may be followed (more details in 11.2.2).

- *Backward recovery* consists in bringing the system back to a past state known to be correct. This is a generally applicable technique, which relies on the periodic saving of the (correct) system state (*checkpointing*).

- *Forward recovery* aims at reconstructing a correct state from the erroneous one. This technique implies that the system state contains enough redundancy to allow reconstruction, which makes it dependent from the specificities of the system. State reconstruction may be imperfect, in which case the service may be degraded.

As a practical conclusion of this brief review, there is no single universal method to ensure fault tolerance. The method chosen for each concrete case strongly depends on its specific aspects. Therefore, it is important to make all assumptions explicit, and to verify that the assumptions are indeed valid. For instance, fault masking by redundant processing is only effective if the replicated elements are subject to independent faults. Modularity is helpful for fault tolerance, since it restricts the range of analysis, and makes error detection and confinement easier.

Achieving fault tolerance remains a difficult proposition, as illustrated by such "horror stories" as the Ariane 5 [Lions et al. 1996] and Therac 25 [Leveson and Turner 1993] failures.

11.2.1 Error Detection

The goal of error detection is to identify the presence of an error, i.e., a system state that deviates from its specification. The motivation is to prevent (if possible) the error to cause the affected component(s) to fail, to avoid the propagation of the error to other parts of the system, and to better understand the fault that caused the error, in order to prevent further occurrences of the fault.

Two qualities are attached to error detection.

- *Latency*, the time interval between the occurrence of an error and its detection.
- *Coverage*, the ratio of detected errors to existing errors.

Error detection techniques depend on the nature of the error. For instance a fault in a processor may cause it to deliver incorrect output, or to stop. In the first case, the output needs to be analyzed by one of the techniques described below; in the second case, a failure has occurred, and failure detection techniques are needed. Failure detection is the subject of 11.4.1. Here we only examine the detection of errors in data.

A first category of methods uses redundancy. One common technique for detecting errors in data transmission relies on error-detecting codes. For example, one extra (parity) bit detects an error affecting one data bit. With more redundancy (error-correcting codes), an error may even be corrected. A textbook on error correction is [Blahut 2003].

For data production (as opposed to transmission), errors may be detected by comparing the results of replicated data sources. For example, bit by bit comparison of the output of two identical processors, with identical inputs, allows error detection. One needs to make sure that the replicated data sources have independent failure modes (e.g., separate power supply, etc.). Techniques based on comparison have a high cost, but ensure low latency and high coverage rate.

Another class of methods relies on plausibility checks. These methods are application dependent. For instance, the values output by a process may be known to be in a certain range, and this property may be checked. Alternatively, bounds may be known for the variation rate in a series of output values. Plausibility checks are usually lest costly than techniques based on redundancy. Their latency depends on the specific situation. Their weak point is their coverage rate, which is often small, because plausibility usually entails loose constraints.

Examples of the use of error detection techniques are presented in the case studies (11.9).

11.2.2 Error Recovery

As noted in 11.2, backward recovery is a general technique, while forward recovery is application-specific. In this section, we briefly examine the main aspects of backward recovery.

Stated in general terms, the principle of backward recovery is to replace an erroneous system state by a previously recorded correct state, an operation called *rollback*. The process of state recording is known as *checkpointing*. The state is copied on stable storage, a highly available storage medium.

The state of a distributed system consists of the local states of its components, together with the state of the communication channels, i.e., the messages that were sent but not yet delivered. This state must be *consistent*, i.e., it must be the result of an observation that does not violate causality. To explain this notion, consider a system made up of two components A and B, and suppose A sends a message m to B. If A registers its state *before* sending m, and B registers its state *after* receiving m, the global checkpointed state is inconsistent, because it records the receipt of a message that has not been sent.

There are two main techniques to construct a consistent state. In coordinated checkpointing, the components of the system register their state in a coordinated way, by exchanging messages, so as to ensure that the recorded state (including the state of the communication channels) is consistent. A coordination algorithm is described in [Chandy and Lamport 1985]. In uncoordinated checkpointing, components checkpoint their state independently. If recovery is needed, a consistent state must be reconstructed from these independent records, which implies that multiple records must be conserved for each component.

Coordinated checkpointing is usually preferred, since it implies less frequent state saving (state recording on stable storage tends to be the major component of the cost). In addition, a technique known as message logging allows a more recent state to be reconstructed from a consistent checkpoint, by recording the messages sent by each process, and replaying the receipt of these messages, starting from the recorded state. A survey of checkpoint-based recovery is [Elnozahy et al. 2002].

Backward recovery is illustrated by two examples, which are further detailed in this chapter.

- *Process pairs*, a technique introduced in the HP NonStop system (originally Tandem [Bartlett 1981]). In this system, all hardware components (CPU, bus, memory modules, controllers, disks) are replicated, and the operating system implements highly available processes, by representing a process by a pair of processes running on different processors. More details in 11.6.2.

- *Micro-reboot*, a technique used in multi-tier middleware [Candea et al. 2004]. This technique is based on the empirical observation that a software failure of undetermined origin is usually cured by rebooting the system (restarting from a "clean" state). The difficulty is to determine the environment to be rebooted: it should be large enough to repair the error, but as small as possible for efficiency. More details in **??**.

See [Bartlett and Spainhower 2004] for a review and a comparison of two highly available commercial systems using recovery (HP-Tandem NonStop and IBM zSeries).

11.2.3 Fault Masking

Fault masking (providing the system with enough internal redundancy to suppress the effect of a fault) has been used for a long time to improve the availability of circuits, through the technique of Triple Modular Redundancy (TMR). The circuit is organized in successive stages (the output of stage i is the input of stage $i + 1$), and the components of each stage are replicated in three instances, connected to a majority voter. The main assumption is that the replicated components are highly reliable and fail independently[3]. The voter compares the output of the three components. In the absence of failures, the three outputs are identical. If one output differs from the other two, it is considered faulty, and discarded. The probability of the three outputs being different (i.e., at least two faulty components) is assumed to be negligible. Since the voter itself is a component that may fail, it is also replicated in three instances. This system tolerates the failure of one component at each stage.

Another instance of hardware fault masking using voting is the design of the Tandem Integrity system, in which the voter compares the output of three replicas of the CPU, thus masking the failure of one replica. In contrast with the Tandem NonStop system (11.2.2), which relies on process pairs supported by a special purpose operating system, Tandem Integrity can use any standard system, such as Unix.

A general method for ensuring fault masking is described in 11.3.1.

11.3 State Machines and Group Communication

A generic model that captures the notion of redundancy, based on state machines, is described in 11.3.1. To use this model to actually implement fault tolerant systems, one needs group communication protocols, whose specification and implementation are respectively examined in 11.3.2 and 11.4.

11.3.1 The State Machine Model

A general model for a hardware or software system (or for a system component) is that of a *deterministic state machine* (SM), executed by a process which receives requests on an input channel and sends answers on an output channel. A number of such machines may be assembled to make up a complex system.

The behavior of the state machine may be described as follows.

Call S_0 the initial state of the SM and S_k its state after having processed the k^{th} request r_k ($k = 1, \dots$). The effect of the receipt of request r_i is the following:

(a) $S_i = F(S_{i-1}, r_i)$

(b) $a_i = G(S_{i-1}, r_i)$

(c) the answer a_i is sent on the output channel.

The functions F and G (the state transition function and the output function, respectively) define the behavior of the SM.

In order to ensure consistency, causal order[4] must be preserved for request transmission: if $send(r)$ and $send(r')$ are the events of sending two requests, and if $send(r)$ happens before $send(r')$, then r' cannot be delivered to the SM unless r has been delivered.

A system implemented as an SM can be made tolerant to a number f of failures, by the following method. The SM (together with the process that executes it) is replicated

[3] While this assumption is usually justified for hardware components, it needs to be reconsidered if the failures are due to environmental conditions, such as high temperature, shared by all replicated components.

[4] Causal order is defined by the *happened before* relationship [Lamport 1978b], which subsumes local order on a single site and "send before receive" order for message transmission between sites.

in N copies (or replicas), where $N = f + 1$ in the case of fail-stop failures, and $N = 2f + 1$ in the case of Byzantine failures (in this latter case, a majority of correct processes is needed). There are two main approaches to ensure that the system is available, as long as the number of faulty replicas is less or equal to f.

- *Coordinator-based replication.* In this approach, also called *primary-backup* [Schneider 1993], a particular replica is chosen as coordinator (or primary); the other replicas are called back-ups. The requests are sent to the primary. The primary's task is (i) to process the requests and to send the replies; and (ii) to keep the back-ups consistent. If the primary fails, one of the back-ups becomes the new primary. As shown in 11.6, this technique relies on a particular communication protocol (reliable view-synchronous broadcast), which guarantees that, whenever the primary fails, the new primary is in a state that allows it to correctly fulfill its function.
- *Active replication.* In this approach, also called *replicated state machine* [Lamport 1978a, Schneider 1990], all replicas have the same role, and each request is directly sent to each replica. Since the behavior of the SM is deterministic, a sufficient condition for the N replicas to behave identically (i.e., to keep the same state and to send the same replies) is that *all replicas receive the same requests, in the same order*. Achieving fault tolerance with active replication therefore relies on a particular communication protocol (causal totally ordered broadcast), which is discussed in more detail in 11.3.2 and 11.4.2, together with other forms of group communication.

These approaches define two basic patterns that are recurrent in fault tolerance techniques. System implementations based on both approaches are described in more detail in 11.6.

11.3.2 Group Communication

Since fault tolerance is based on replication, process groups play a central role in the design of fault tolerant systems. For example, as seen in 11.2.3, a group of processes can be organized as a reliable implementation of a single process, using the replicated state machine approach. However, this places specific requirements on communication within the group (in this case, totally ordered broadcast is required).

In this section, we introduce the main concepts and terminology of process groups and we discuss the requirements of group communication and membership protocols. The implementation of these protocols is subject to impossibility results, which are examined in 11.3.3. Practical approaches to group communication are the subject of 11.4.

A *process group* is a specified set of processes (the members of the group), together with protocols related to:

- Group communication (broadcast or multicast), i.e., sending a message to the members of the group, with specified requirements.
- Group membership, i.e., changing the composition of the group. The system maintains knowledge about the current composition of the group, represented by a *view* (a list of processes).

A global picture of group protocols is shown in Figure 11.3.

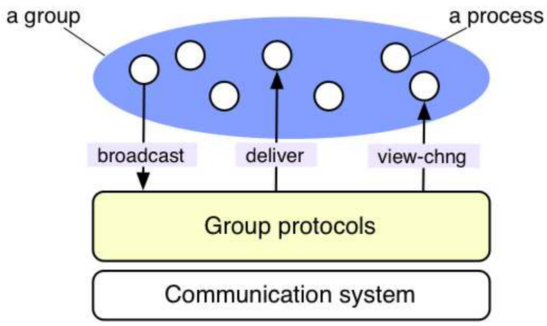

Figure 11.3. Group protocols

A typical API for group protocols includes the following primitives:

- $broadcast(p, m)$. Process p broadcasts message m to the group.
- $deliver(p, m)$. Message m is delivered to process p.
- *view-chng*(p, id, V). Following a change in the group membership, a new view, V, identified by number id, is delivered to process p.

Additional primitives may be provided by specific systems (examples are given below).

Process groups are useful for implementing fault tolerant systems, and also for supporting information sharing and collaborative work.

Group protocols are an important subject of ongoing research, because two of their main aspects are still not well understood. First, specifying a group protocol is a non-trivial task. Group protocol specification is the subject of [Chockler et al. 2001], who note that most existing specifications (at the date of writing) are incomplete, incorrect, or ambiguous. Second, implementing fault tolerant group protocols is often algorithmically difficult, or, in some cases, impossible (see 11.3.3).

We now review the specifications of the most usual protocols.

Group communication protocols

In group communication primitives, a process (the *sender*) sends a message to a set of destination processes (the *receivers*). There are two main forms of group communication primitives, which differ by the definition of the receivers.

- *Broadcast.* The receivers are the processes of a single process set, which may be explicitly or implicitly defined, and which includes the sender. Examples: all members of a process group; "all" processes of a system.
- *Multicast.* The receivers are the members of one or several process groups, which may or not overlap. The sender may or not be part of the receivers.

The main problems specific to multicast are related to overlapping destination groups. We only consider here the case of broadcast.

The specifications of broadcast are discussed in detail in [Hadzilacos and Toueg 1993] and [Chockler et al. 2001]. Assume a fail-stop failure mode (without repair) for processes, and a reliable, asynchronous communication system. A process is said to be *correct* if it does not fail.

The minimal requirement for a broadcast primitive is that it be *reliable*, which is an "all or nothing" property: a message must be delivered to all of its correct receivers, or to none. More precisely, if a message is delivered to *one* correct receiver, it must be delivered to *all* correct receivers. A broadcast protocol that is not reliable is not of much practical use. The value of reliability lies in the shared knowledge that it implies: when a broadcast message is reliably delivered to a correct process, the process "knows" that this message will be delivered to all correct processes in the destination set.

A much stronger requirement is that of total order. A *totally ordered*, or *atomic*, broadcast is a reliable broadcast in which all receivers get the messages in the same order. More precisely, if two messages, m_1 and m_2, are delivered to a correct process p in that order, then m_2 may not be delivered to another correct process q unless m_1 has been first delivered to q.

The two above requirements are independent of the order in which messages are being sent. Another set of requirements involves the sending order:

FIFO broadcast: two messages issued by the same sender must be delivered to all receivers in their sending order. More precisely, if a process has broadcast m_1 before m_2, then m_2 may not be delivered to a correct process q unless m_1 has been first delivered to q.

Causal broadcast: If the sending events of two messages are causally ordered (see 11.2.3), then the messages must be delivered to any receiver in their causal order. More precisely, if the sending of m_1 happened before the sending of m_2, then m_2 may not be delivered to a correct process q unless m_1 has been first delivered to q. FIFO is a special case of causal broadcast.

These requirements are orthogonal to those of reliability and atomicity, which leads to the main six forms of broadcast summarized on Figure 11.4.

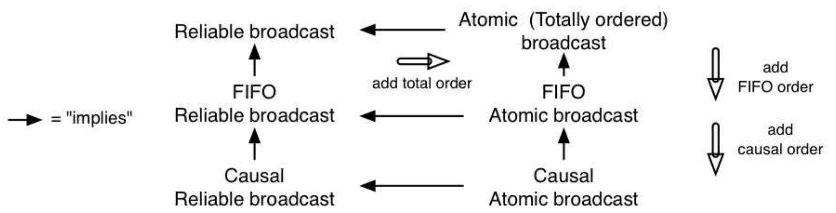

Figure 11.4. Requirements for broadcast protocols (adapted from [Hadzilacos and Toueg 1993])

An additional requirement, again orthogonal to the previous ones, is uniformity. In a *uniform* broadcast, we are also interested in the behavior of faulty processes. For example, reliable uniform broadcast is specified as follows: if a message m is delivered to a (correct or faulty) process, then it must be delivered to all correct processes. The motivation behind this requirement is that the delivery of a message may trigger an irreversible action, and a process may get a message and perform that action before failing.

The above specification assume that the group of processes is static, i.e., processes cannot join or leave the group (they may only crash). The specification may be extended to dynamic groups [Schiper 2006a], using the group membership service defined below.

Group membership protocols

Recall that a group (in the present context) is a set of processes, the members of the group, together with an API for communication and membership. We now assume that the composition of the group changes over time: members may leave the group, or fail; new processes can join the group, and failed processes can be repaired and join the group again. The aim of group membership protocols is to implement the join and leave operations, and to keep the members of the group informed about the current membership, by delivering a sequence of views.

There are two versions of group membership protocols. The *primary partition* version assumes that the sequence of views is totally ordered. This is the case if the process group remains connected (i.e., all its members can communicate with each other), or if one only considers a single partition in a disconnected group. In the *partitionable* version, the sequence of views is partially ordered. This applies to a partitioned group, in which several partitions are taken into account. We only consider the primary partition version.

As mentioned above, writing precise and consistent group membership protocol specifications is a surprisingly difficult task. Recall (1.4.2) that the required properties are categorized into *safety* (informally: no undesirable event or condition will ever occur) and *liveness* (informally: a desirable event or condition will eventually occur). These requirements are summarized as follows (see [Chockler et al. 2001] for formal statements).

Recall that the group membership service delivers views to the members of the group. A *view* is a list of processes that the service considers to be current members of the group. When a view v is delivered to process p, p is said to *install* the view.

The safety properties include:

- Validity. A process which installs a view is part of that view (self-inclusion property).
- Total order. The set of views installed by the processes is totally ordered (thus one may speak of the next view, of consecutive views, etc.).
- Initial view. There exists an initial view, whose members are explicitly defined.
- Agreement. The sequence of views installed by a process is a subsequence of *contiguous* elements of the global sequence of views (consistency property).
- Justification. A change of view (installing a new view) must be motivated by one of the following events: a process has joined the group, left the group, or failed.
- State transfer. The state of the group (i.e., the contents of a view) may be transferred to the next view in the sequence, except if all processes have failed. "State transfer" means that at least one process that installed the new view was included in the predecessor of that view.

Liveness is defined by the following property: any event associated with a given process (join, leave, or failure) will eventually be visible in a further view (except if a process fails after joining, in which case it may never install a view).

A recent proposal [Schiper and Toueg 2006] suggests to consider group membership as a special case of a more general problem, set membership, in which the processes of the group may add or remove elements of a set, and need to agree on the current composition of the set. The members of the set are drawn from an arbitrary universe. Group membership is the special case in which the members of the set are processes. This approach favors separation of concerns (1.4.2), by identifying two separate issues: determining the set of processes that are deemed to be operational, and ensuring agreement over the successive values of that set. Different algorithms are needed to solve these two problems.

Virtual synchrony

One important aspect of group protocols is the connection between group membership and communication through the notion of *virtual synchrony*, first introduced in [Birman and Joseph 1987]. Virtual synchrony defines a broadcast protocol (view synchronous broadcast) that is consistent with group membership, as implemented by the view mechanism.

The specification of view synchronous broadcast extends the agreement condition defined for group membership. The extended conditions are as follows.

- *Agreement* for view synchronous communication. Two properties hold:
  - The (correct) members of the process group install the same sequence of totally ordered views of the group.
  - The (correct) members of the process group see the same sequence of delivered messages between the installation of a view v_i and that of the next view v_{i+1}.
- *Justification* for view synchronous communication: any message delivered has actually been sent by some process .
- *Liveness* for view synchronous communication: any message sent by a correct process is eventually delivered.

These conditions are illustrated by the examples shown in Figure 11.5. In all cases, process p_1 broadcasts a message to a group including itself and p_2, p_3, p_4, all alive in view v_i. In case (a), there is no view change, and the system behaves correctly. In case (b), p_1 crashes during broadcast, and the message is delivered to p_3 and p_4 but not to $p2$. This violates virtual synchrony, since the surviving processes in view v_{i+1} have not received the same set of messages. This also happens in case (c), for a different reason: although the message eventually gets to p_4, it is only delivered after p_4 has installed the new view; p_4 is inconsistent in the meantime. Finally, virtual synchrony is respected in case (d), since the surviving processes in view v_{i+1} have received the same set of messages when the view is installed.

One common use of virtual synchrony is illustrated by the example of a replicated database. Each replica is under the control of a manager process, and updates are (atomically) broadcast to all managers. If this broadcast does not respect virtual synchrony,

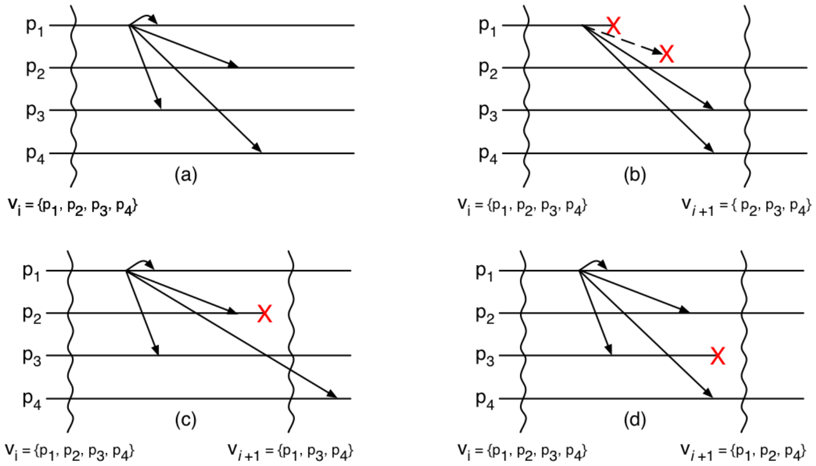

Figure 11.5. Virtual synchrony

one ore more replica(s) will become inconsistent after a view change following the crash of one of the managers. Virtual synchrony ensures that a reader can consult any of the replicas; it may get an out of date version (one that has not yet received the last update), but not an inconsistent one. Data replication is discussed in more detail in 11.7. Another application of virtual synchrony (fault tolerant servers) is presented in 11.6.2.

There is a close relationship between group membership and group communication. Actually, each of these services can be implemented on top of the other one. In most usual implementations, the group membership layer lies beneath the group communication protocol stacks. In [Schiper 2006a], a different approach is proposed, in which atomic broadcast is the basic layer, and group membership is implemented above this layer. The claimed advantage is simplicity, and better efficiency in most practical situations.

Both group membership and atomic broadcast can be expressed in terms of an agreement protocol, consensus, which we now examine.

11.3.3 Consensus and Related Problems

Consensus is a basic mechanism for achieving agreement among a set of processes. It can be shown that both the total order broadcast and the virtually synchronous group membership protocols are equivalent to consensus. The specifications and implementations of consensus therefore play a central role in the understanding of fault tolerant distributed algorithms. It has been proposed [Guerraoui and Schiper 2001] that a generic consensus service be provided as a base for building group membership and group communication protocols.

Consensus Specification

Given a set of processes $\{p_i\}$ linked by a communication system, consensus is specified as follows. Initially, each process p_i proposes a value v_i. If the algorithm terminates, each process p_i decides a value d_i. The following conditions must hold:

- *Agreement.* No two correct processes decide different values.
- *Integrity.* Each process decides at most once (i.e., if it decides, the decision may not be modified).
- *Validity.* A decided value must be one of those proposed (this condition eliminates trivial solutions, in which the processes decide some predetermined value).
- *Termination.* If at least one correct process starts the algorithm, all correct processes decide in finite time.

A process is correct if it does not fail.

Solving consensus in the absence of failures is a simple task. It may be done in two ways, corresponding to the patterns identified in 11.3.1.

1. One process is defined as the coordinator (or *primary*). On the primary's request, each process sends its proposed value to the primary. After it has received all values, the primary chooses one of them as the decided value, and sends it to all processes. When it receives a value from the primary, each process decides that value.
2. Each process sends its proposed value to every process (including itself). Once a process has received proposed values from all the p_is, it applies a deterministic decision algorithm (the same one for all processes) to choose a value among those received, and it decides that value.

In both cases, if we assume reliable communication, the algorithm terminates. This is true even if communication is asynchronous, although the termination time is not bounded in that case.

The two above approaches are the base of the algorithms used to solve consensus in the presence of failures. In that case, achieving consensus becomes much more complex, and in many cases intractable, as discussed below.

We now present (without proof) a few important results on the tractability of the problems associated with group protocols. The main impossibility results concern asynchronous systems and are linked to the impossibility of "perfect" failure detection in such systems. We then discuss the main approaches to the solution of agreement problems, which are the base of group protocols.

Limits of Group Protocol Algorithms

Reliable broadcast can be implemented in an asynchronous system, using a "flooding" algorithm. When a process receives a message, it transmits the message to all its neighbors (the processes to which it is connected by a direct link), and then delivers the message to itself. To broadcast a message, the sender sends it to itself, and follows the above rule. This algorithm implements the specification of reliable broadcast; in addition, it is uniform, i.e., if a message is delivered to a (correct or faulty) process, then it is delivered to all correct processes. Uniformity is guaranteed by the fact that a process delivers a message to itself only *after* having sent it to its neighbors. Since the system is asynchronous, there is no upper bound on the delivery time of a message.

This algorithm can be extended to achieve FIFO and causal reliable broadcast [Hadzilacos and Toueg 1993].

The central impossibility result, often known as FLP from its authors' names Fischer, Lynch, and Paterson [Fischer et al. 1985], is the following: in an asynchronous system, there is no deterministic algorithm that achieves consensus in finite time if one process (at least) may fail. Recall the assumptions: the failure mode is fail-stop, and communication is reliable. This impossibility extends to totally ordered broadcast and to view-synchronous group membership, since these algorithms are equivalent to consensus.

With the same assumptions, it has been demonstrated [Chandra et al. 1996b] that group membership cannot be solved in an asynchronous system.

How to live with these limitations? Three main tracks have been explored.

- Use randomization. The FLP result only applies to deterministic algorithms. Randomized algorithms have been devised to achieve consensus in an asynchronous system (see a review in [Aspnes 2003]).
- Relax the asynchrony assumption. Various forms of "partial synchrony" have been investigated [Dwork et al. 1988]. We use that specified in 11.1.4.
- Use an imperfect algorithm, i.e., one that may fail, but which is the best possible in some sense. Two main approaches have been proposed: the Paxos algorithm, and unreliable failure detectors. They are examined below.

Most practical approaches to group protocols combine the use of one of the above algorithms and a partial synchrony assumption. They are discussed in Section 11.4.

The Paxos consensus protocol

The first approach to solving consensus is known as the Paxos protocol. The failure hypotheses are weaker than indicated above: the network may lose messages, and the processes may fail by crashing, and subsequently recover. Paxos was introduced in [Lamport 1998]; an equivalent protocol was included in the replicated storage system of [Oki and Liskov 1988]. A pedagogical description may be found in [Lampson 2001].

The protocol works roughly as follows. The system goes through a sequence of views. A *view* defines a period during which one selected process, the primary, attempts to achieve agreement (using the same principle as the primary-based algorithm described in 11.3.2 in the absence of failures). After a view is started as the result of a view change, the protocol works in two phases. In the first phase, the primary queries the participants for their state (i.e., their replies in past views). Then the primary selects an "acceptable" value (in a sense defined later), and proposes that value[5]. If a majority agrees on that value, consensus is achieved, and the decision is propagated to the participants. Thus the protocol succeeds if the primary and a majority of processes live for a sufficiently long time to allow two and a half rounds of exchanges between the primary and the other processes. If either a majority cannot be gathered or the primary fails before agreement is reached or progress

[5] When the protocol is started by a client' request (as opposed to a view change), the value is that proposed by the client

is "too slow", a new primary is elected[6], and a new view is started. The view change protocol is the choice of a new primary, followed by the first phase described above. The "normal" protocol achieves safety (agreement on a value), while the view change protocol guarantees liveness (i.e., progression).

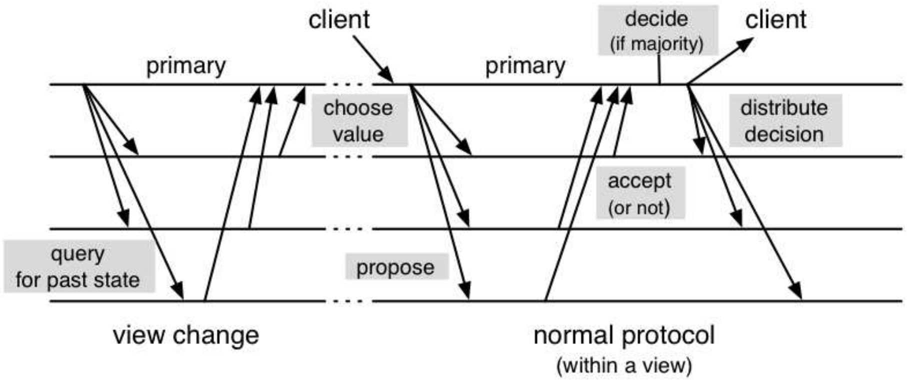

Figure 11.6. The Paxos consensus protocol

Paxos does not require that a single primary should be present at a time. Several views (and therefore several primaries) may coexist, depending on how the election process was started upon detecting failure to agree. However, Paxos imposes a total order on views, allowing processes to reject the values proposed by any primary that is not in the most recent view. A key point is that Paxos restricts the primary's choice of acceptable values: if a majority has once agreed on a value in some view (and a decision has not been reached in that view because of failures), only that value may be proposed in subsequent views. This stability property ensures that no two different values may be decided, a safety property.

The actual working of Paxos is much more subtle than shown in the above rough description (see references above). In addition, a number of optimizations may be made, which complexify the description. Paxos is not guaranteed to ensure agreement in finite time (since FLP still holds), but it is extremely robust in the presence of failures and unknown delays.

One practical use of Paxos is consistency management in replicated storage systems (11.7). See [Oki and Liskov 1988] for an early application, and [Chandra et al. 2007] for a recent one.

Unreliable failure detectors

The idea of *unreliable* failure detectors [Chandra and Toueg 1996] is based on the following intuitive remark: the intractability of consensus in asynchronous systems is linked to the impossibility of telling a faulty process from a slow one, i.e., to the impossibility of building a "perfect" failure detector. This leads to the following questions: (i) what are the properties of a perfect failure detector? and (ii) is consensus solvable with an "imperfect" failure detector, and, if so, what properties are required from such a detector?

[6]The election protocol may be very simple, as long as it selects one primary per view. Conflicts between coexisting views are solved as explain later.

A *failure detector* is a device that can be used as an oracle: it delivers upon request the list of processes that it suspects of being faulty. In a distributed system, a failure detector is itself a distributed program, made up of cooperating components, one on each site. To use the detector, a process consults the local failure detector component.

Two properties have been identified for a *perfect* failure detector: *strong completeness*, i.e., every faulty process is eventually suspected by every correct process; and *strong accuracy*, i.e., no correct process is ever suspected by any correct process. Thus, there is a time after which the list delivered by a perfect detector contains all the faulty processes, and only those processes.

These properties can be weakened[7], allowing several classes of imperfect (or unreliable) detectors to be defined. Weak accuracy means that *some* correct process is never suspected by any correct process. Both strong and weak accuracy may be *eventual*; in that case, there exists a time after which the indicated property holds.

One may then define the following classes[8] of detectors by their properties:

- *Perfect* (P): strong completeness, strong accuracy.
- *Strong* (S): strong completeness, weak accuracy.
- *Eventually Perfect* ($\diamond P$): strong completeness, eventually strong accuracy.
- *Eventually Strong* ($\diamond S$): strong completeness, eventually weak accuracy.

The detectors other than P are unreliable, i.e., they can make false suspicions.

The most important result [Chandra and Toueg 1996] is the following: in an asynchronous system with fail-stop failures, consensus can be solved using a detector in any of the classes P, S, $\diamond P$, $\diamond S$. Moreover [Chandra et al. 1996a], $\diamond S$ is the weakest detector allowing consensus.

More precisely, the solutions to the consensus problem among n processes, using failure detectors, have the following properties.

- With P: allows up to $n-1$ failures, needs $f+1$ rounds, where f is the number of failures tolerated.
- With S: allows up to $n-1$ failures, needs n rounds.
- With $\diamond S$: allows up to $\lceil n/2 \rceil - 1$ failures, needs a finite (but unbounded) number of rounds.

The consensus algorithm for a $\diamond S$ detector is based on the principle of the *rotating coordinator*. The coordinator tries to achieve agreement among a majority of processes, based on the proposed values. This may fail, because the coordinator may be falsely suspected or may crash. The function of coordinator rotates among the processes. Eventually (as guaranteed by the properties of $\diamond S$), a correct process that is never suspected

[7]We do not need to consider weak completeness (every faulty process is eventually suspected by *some* correct process), because a weakly complete detector is easily converted into a strongly complete one, by making each process reliably broadcast the list of suspected processes.

[8]A detector belongs to a given class if it has the properties specified for that class. In the following, we often use the term "detector C" to denote a detector of class C, where C is one of P, ... $\diamond S$.

will become coordinator, and will achieve agreement, provided that a majority of correct processes exists.

None of the above detectors may be implemented in an asynchronous system by a deterministic algorithm (if this were the case, it would contradict the FLP result). Implementations under a partial synchrony assumption are described in 11.4.1.

Paxos and unreliable failure detectors are two different approaches to solving consensus. A detailed comparison of these approaches is still an open issue. The interest of Paxos is that it ensures a high degree of tolerance to failures, (including message loss), and that it may be extended to Byzantine failures. The interest of unreliable failure detectors is that they allow a deep understanding of the origins of the difficulty of achieving consensus, and that they provide guidelines for actual implementations (see 11.4.1).

11.4 Practical Aspects of Group Communication

In this section, we examine the actual implementation of group protocols. We start by discussing implementation issues for failure detectors (11.4.1). We then look at atomic broadcast (11.4.2) and group membership (11.4.3). We conclude with a discussion of group protocols for large scale systems (11.4.4).

11.4.1 Implementing Failure Detectors

We first present the basic assumptions and mechanisms that underlie the implementations of failure detectors. We then discuss the quality factors of these detectors, and conclude with a few practical considerations.

Basic Mechanisms of Failure Detection

Recall (11.3.3) that none of the failure detectors P, S, $\diamond P$, $\diamond S$ may be implemented by a deterministic algorithm in an asynchronous system. The approach usually taken in practice is to relax the asynchrony assumption, by considering a particular form of *partial synchrony*: assume that bounds exist for the message transmission time and for the speed ratio between processes, but that these bounds are unknown *a priori*, and hold only after a certain time. This assumption is realistic enough in most common situations, and it allows the use of timeouts for implementing failure detectors. Recall that the failure model for processes is fail-stop, without recovery.

Failure detection relies on an elementary mechanism that allows a process q to determine if a process p is correct or faulty, assuming an estimated upper bound $\Delta_{p,q}$ is known for the transmission time[9] of a message between p and q. Two variants of this mechanism have been proposed (Figure 11.7).

- *Pull*, also called *ping*. Process q periodically sends a request "are you alive?" to process p, which answers with a message "I am alive". If no answer is received after $2\Delta_{p,q}$, q suspects p of having crashed.

[9] also assume, for simplicity, that $\Delta_{p,q} = \Delta_{q,p}$

- *Push*, also called *heartbeat*. Process p periodically sends to q (and possibly to other processes) a message "I am alive". Assume that p and q have synchronized clocks, and that q knows the times t_i at which p sends its message. If, for some i, q has not received the heartbeat message from p by time $t_i + \Delta_{p,q}$, q suspects p of having crashed.

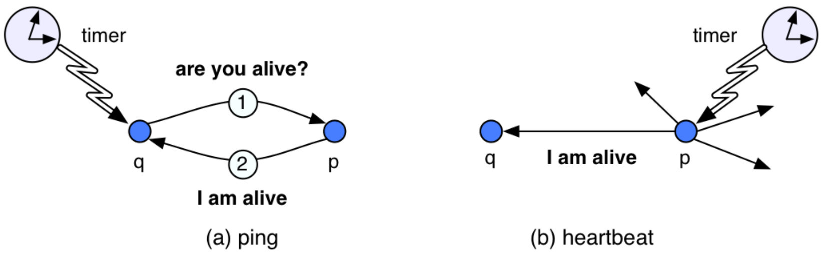

Figure 11.7. Principle of ping and heartbeat

The above description only gives the general principle, and various details need to be filled in. In particular, how is $\Delta_{p,q}$ estimated? One common technique is to dynamically determine successive approximations to this upper bound for each process, by starting with a preset initial value. If the value is found to be too small (because a heartbeat message arrives after the deadline), it is increased (and the falsely suspected process is removed from the suspects' list). With the partial synchrony assumption, this technique ensures convergence towards the eventual upper bound.

Another important design choice for a failure detection algorithms is the communication pattern between the processes. Two main patterns have been considered.

- Complete exchange (or all-to-all communication). Every process communicates with every other one. For n processes, the number of messages is thus of the order of n^2 per run (more precisely nC, where C is the number of faulty processes in the considered run).

- Exchange over a logical (or virtual) ring. Each process only communicates with some[10] of his successors or predecessors on the ring. The number of messages is now linear in n.

In addition, hierarchical patterns (with separate intra-group and inter-group communication) have been introduced for large scale systems (see 11.4.4).

Various combinations of the design choices indicated above (ping or heartbeat, determining the starting point for the watchdog timers, estimating an upper bound for the communication delay, choosing a communication pattern) have been proposed. A summary of some proposals follows.

In their seminal paper on unreliable failure detectors, [Chandra and Toueg 1996] propose an implementation of a failure detector $\diamond P$, assuming partial synchrony (as specified

[10] Typically, if an upper bound f is known for the number of tolerated failures, a process needs to communicate with its $f + 1$ successors to ensure it will reach a correct process.

at the beginning of this section). This implementation uses heartbeat, all-to-all communication, and adaptive adjustment of the transmission time bound, as described above. The main drawback of this implementation is its high communication cost (quadratic in the number of processes).

To reduce the number of messages, [Larrea et al. 2004] propose implementations of $\diamond P$ and $\diamond S$ using a logical ring, a ping scheme for failure detection, and again an adaptive adjustment of the transmission time bound. Each process q monitors (i.e., "pings") a process p, called the "target" of q; initially, this target is $succ(q)$, the successor of q on the ring. If q suspects p, it adds it to its suspects lists, and its new target becomes $succ(p)$; if q stops suspecting a process r, it also stops suspecting all processes between r and its current target, and r becomes its new target. The drawback of this algorithm is that each process only maintains a "local" (partial) suspects list. To build the global list (the union of all local lists), each local list needs to be broadcast to all processes. While this transmission may be optimized by piggybacking the local list on top of ping messages, this delays the detection, by a process q, of faulty processes outside q's own local list. Several variations of this scheme, still based on a logical ring but using heartbeat instead of ping, are proposed in [Larrea et al. 2007].

[Mostefaoui et al. 2003] take a different approach, by changing the assumptions on the communication system. They do not assume partial synchrony, but introduce assumptions involving an upper bound f on the number of faulty processes. These assumptions amount to saying that there is at least one correct process that can monitor some process (i.e., receive responses to its ping messages to that process).

There does not appear to exist a "best" failure detector, due to the wide range of situations, and to the variety of cost and quality metrics. The quality metrics are discussed below.

Quality Factors of Failure Detectors

A comprehensive investigation of the quality of service (QoS) factors of failure detectors is presented in [Chen et al. 2002]. This work was motivated by the following remarks:

- The properties of failure detectors (11.3.3) are specified as eventual. While this quality ensures long term convergence, it is inappropriate for applications that have timing constraints. The *speed* of detection is a relevant factor in such situations.

- By their nature, unreliable failure detectors make false suspicions. In practical situations, one may need a good *accuracy* (reducing the frequency and duration of such mistakes).

[Chen et al. 2002] propose three primary QoS metrics for failure detectors. The first one is related to speed, while the other two are related to accuracy. They also define additional quality metrics, which may be derived from the three primary ones.

The primary metrics are defined as follows, in the context described at the beginning of 11.4.1, i.e., a system composed of two processes p and q, in which q monitors the behavior of p, and q does not crash (we only give informal definitions; refer to the original paper for precise specification).

- Detection time. This is the time elapsed between p's crash and the instant when q starts suspecting p permanently.
- Mistake recurrence time. This is the time between two consecutive mistakes (a mistake occurs when q starts falsely suspecting p). This is analogous to the notion of $MTTF$ as defined in 11.1.1 (although the "failure" here is a faulty behavior of the detector).
- Mistake duration. This is the time that it takes to the failure detector to correct a mistake, i.e., to cease falsely suspecting a correct process. This is analogous to the notion of $MTTR$ as defined in 11.1.1.

Since the behavior of the system is probabilistic, the above factors are random variables. More precisely, they are defined by assuming the system has reached a steady state, i.e., a state in which the influence of the initial conditions has disappeared[11]. Note that these metrics are implementation-independent: they do not refer to a specific mechanism for failure detection.

[Chen et al. 2002] present a heartbeat-based algorithm, and show that it can be configured to reach a specified QoS (as defined by the above primary factors), if the behavior of the communication system is known in terms of probability distributions for message loss and transmission time. The algorithm makes a best effort to satisfy the QoS requirements, and detects situations in which these requirements cannot be met. Configuration consists in choosing values for the two parameters of the detector: η, the time between two successive heartbeats, and δ, the time shift between the instants at which p sends heartbeats and the latest instants[12] at which q expects to receive them before starting to suspect p. If the behavior of the communication system varies over time, the algorithm may be made adaptive by periodically reevaluating the probability distributions and re-executing the configuration algorithm.

Concluding Remarks

The results discussed above rely on several assumptions. Are these assumptions valid in practice?

Concerning the partial synchrony assumption, observation shows that a common behavior for a communication system alternates between long "stable" phases, in which there is a known upper bound for transmission times, and shorter "unstable" phases, in which transmission times are erratic. This behavior is captured by the partial synchrony model, since an upper bound for the transmission time eventually holds at the end of an unstable phase.

Concerning reliable communication, the "no message loss" assumption is justified for LANs and WANs, in which message retransmission is implemented in the lower layer protocols. It is questionable in mobile wireless networks.

[11] In practice, steady state is usually reached quickly, typically after the receipt of the first heartbeat message in a heartbeat-based detector

[12] The basic algorithm assumes that the clocks of p and q are synchronized; it can be extended, using estimation based on past heartbeats, to the case of non-synchronized clocks.

Concerning the fail-stop failure model, the assumption of crash without recovery is overly restrictive. In practice, failed components (hardware or software) are repaired after a failure and reinserted into the system. This behavior can be represented in two ways.

- Using dynamic groups. A recovered component (represented by a process) can join the group under a new identity.
- Extending the failure model. A crash failure model with recovery has been investigated in [Aguilera et al. 2000].

In all cases, recovery protocols assume that a form of reliable storage is available.

Concerning the basic failure detection mechanisms, both heartbeat and ping are used, since they correspond to different tradeoffs between efficiency (in terms of number of messages) and accuracy. Heartbeat seems to be the most common technique. Since ping uses more messages, it is usually associated with a specific organization of the interprocess links, such as a tree or a ring, which reduces the connectivity.

11.4.2 Implementing Atomic Broadcast

A large number of atomic broadcast and multicast algorithms have been published. [Défago et al. 2004] give an extensive survey of these algorithms and propose a taxonomy based on the selection of the entity used to build the order. They identify five classes, as follows.

- *Fixed sequencer.* The order is determined by a predefined process, the sequencer, which receives the messages to be broadcast and resends them to the destination processes, together with a sequence number. The messages are delivered in the order of the sequence numbers. This algorithm is simple, but the sequencer is a bottleneck and a single point of failure.
- *Moving sequencer.* Same as above, but the role of sequencer rotates among the processes. The advantage is to share the load among several processes, at the price of increased complexity.
- *Privilege-based.* This class of algorithms is inspired by the mutual exclusion technique using a privilege in the form of a token, which circulates among the processes. A process can only broadcast a message when it holds the token. To ensure message ordering, the token holds the sequence number of the next message to be broadcast. This technique is not well suited for dynamic groups, since inserting or removing a process involves a restructuring of the circulating pattern of the token. Also, special care is needed to ensure fairness (such as specifying a limit to the time that a process holds the token).
- *Communication History.* The delivery order is determined by the senders, but is implemented by delaying message delivery at the destination processes. Two methods may be used to determine the order: causal time-stamping, using a total order based on the causal order, and deterministic merge of the message streams coming from each process, each message being independently (i.e., non causally) timestamped by its sender.

- *Destinations Agreement*. The delivery order is determined through an agreement protocol between the destination processes. Here again, there are several variants. One variant uses a sequence of consensus runs to reach agreement over the sequence of messages to be delivered. Another variant uses two rounds of time-stamping: a local time-stamp is attached to each message upon receipt, and a global time-stamp is determined as the maximum of all local timestamps, thus ensuring unique ordering (ties between equal timestamps are resolved using process numbers).

In addition, [Défago et al. 2004] describe a few algorithms that are hybrid, i.e., that mix techniques from more than one of the above classes.

Two delicate issues are (i) the techniques used to make the algorithms fault-tolerant; and (ii) the performance evaluation of the algorithms. This latter aspect often relies on local optimization (e.g., using piggybacking to reduce the number of messages, or exploiting the fact that, on a LAN, messages are most often received in the same order by all processes). Therefore, the algorithms are difficult to compare with respect to performance.

Regarding fault tolerance, two aspects need to be considered: failures of the communication system, and failures of the processes.

Many algorithms assume a reliable communication system. Some algorithms tolerate message loss, using either positive or negative acknowledgments. For example, in a fixed sequencer algorithm, a receiving process can detect a "hole" in the sequence numbers, and request the missing messages from the sequencer. As a consequence, the sequencer needs to keep a copy of a message until it knows that the message has been delivered everywhere.

Concerning process failures, most of the proposed algorithms rely on a group membership service, itself based on a failure detector. Others directly use a failure detector, either explicitly, or implicitly through a consensus service. The relationship between atomic broadcast, group membership and consensus is briefly discussed in the next section.

11.4.3 Implementing Virtually Synchronous Group Membership

There are three main approaches to group membership implementation.

- Implementing group membership over a failure detector (itself implemented by one of the techniques described in 11.4.1). This is the most common approach. An example using it is described below.

- Implementing group membership over atomic broadcast (itself implemented by one of the techniques described in 11.4.2). [Schiper 2006a] gives arguments in favor of this approach.

- Implementing both atomic broadcast and group membership over a consensus service (itself usually based on a failure detector). See [Guerraoui and Schiper 2001].

We illustrate the first approach (using a failure detector to implement virtual synchronous group membership) with the example of JavaGroups [Ban 1998], itself derived from the Ensemble protocol stack [van Renesse et al. 1998]. The protocol uses the coordinator pattern and is based on the notion of stable messages. A message is said to be *stable* when it is known to having been delivered to every member of the current view of the group. Otherwise, it is *unstable*[13].

A view change is triggered by one of the following events (11.3.2): a process joins the group, leaves the group, or crashes (this latter event is observed by a failure detector). The coordinator is notified when any of these events occurs. It then initiates a view change by broadcasting a FLUSH message to the group. When a process receives the FLUSH message, it stops sending messages (until it has installed the new view), and sends all the messages it knows to be unstable to the coordinator, followed by a FLUSH_OK message. When it has received all FLUSH_OK replies, the coordinator broadcasts all the unstable messages (messages are uniquely identified within a view, by a sequence number, to avoid duplicates). Recall that the underlying communication system is reliable, so that the messages will reach their destination if the sender is correct. When all messages have become stable, the coordinator broadcasts a VIEW message, which contains the list of the members of the new view. When a process receives this message, it installs the new view.

In order to detect failures, the processes are organized in a logical ring, and each process pings its successor (as described in 11.4.1). If a process (other than the coordinator) crashes during this view change protocol, a notification is sent to the coordinator, which starts a new view change. If the coordinator crashes, its predecessor in the ring (which detected the crash) becomes the new coordinator, and starts a new view change.

The reliable message transmission layer uses both a positive (ACK) and negative (NAK) acknowledgment mechanism. Negative acknowledgment is based on the sequential numbering of messages, and is used when the received sequence number differs from the expected one. For efficiency, the default mode is NAK for ordinary messages. The ACK mode is used for sending view changes (the VIEW message).

This protocol does not scale well (the group size is typically limited to a few tens of processes). A hierarchical structure based on a spanning tree allows a better scalability. The processes are partitioned into local groups; in each local group, a local coordinator (the group controller) broadcasts the flush and view messages within the group. The controllers cooperate to ensure global message diffusion. This allows efficient communication for groups of several hundred processes.

Other communication toolkits include Appia [Appia] and Spread [Spread].

11.4.4 Large Scale Group Protocols

This section still unavailable

11.5 Fault Tolerance Issues for Transactions

11.5.1 Transaction Recovery

This section still unavailable.

[13]Typically, a message that was broadcast by a process that failed during the broadcast may be unstable

11.5.2 Atomic Commitment

This section still unavailable.

11.6 Implementing Available Services

In this section and the next one, we examine how available systems can be built using replication. This section deals with available services, while section 11.7 discusses available data. Both cases involve replicated entities, which raises the issue of keeping these entities mutually consistent. We examine consistency conditions in 11.6.1.

Recall (2.1) that a *service* is a contractually defined behavior, implemented by a component and used by other components. In the context of this discussion, we use the term *server* to designate the component that implements a service (specified by its provided interface), together with the site on which it runs. Making a service highly available is achieved by replicating the server. Two main approaches are used, corresponding to the two patterns identified in 11.3.1, assuming fail-stop failures: the primary-backup protocol (11.6.2) and the active replication protocol (11.6.3). These protocols are discussed in detail in [Guerraoui and Schiper 1997].

The case of Byzantine failures is the subject of 11.6.4.

11.6.1 Consistency Conditions for Replicated Objects

Defining consistency conditions for replicated data is a special case of the more general problem of defining consistency for concurrently accessed data. This problem has been studied in many contexts, from cache coherence in multiprocessors to transactions. Defining a consistency condition implies a trade-off between strong guarantees and efficient implementation. Therefore, a number of consistency models have been proposed, with different degrees of strictness.

Consider a set of shared data that may be concurrently accessed by a set of processes, which perform *operations* on the data. Operations may be defined in different ways, according to te application context. In a shared memory, operations are elementary reads and writes. In a database system, operations are transactions (see Chapter 9). In a (synchronous) client-server system (see Chapter 5), an operation, as viewed by the server, starts when the server receives a client's request, and ends when the server sends the corresponding reply to the client.

A commonly used consistency requirement for concurrently accessed data is *sequential consistency* [Lamport 1979], which specifies that a set of operations, concurrently executed by the processes, has the same effect as if the operations had been executed in some sequential order, compatible with the order seen at each individual process. In the context of transactions, this requirement is usually called *serializability* (9.2.1).

A stronger requirement, known as *linearizability* [Herlihy and Wing 1990], is used for shared objects. A shared object may be seen as a server that receives requests from client processes to perform operations. A partial order is defined on the operations, as follows: operation O_1 precedes operation O_2 (on the server) if the sending of the result of O_1 precedes the receipt of the request of O_2. An execution history (a sequence of operations on the object) is said to be linearizable if (i) it is equivalent to a legal sequential execution (legal means "satisfying the specification of the object"); and (ii) the sequential order of the operations is compatible with their ordering in the initial history. Linearizability thus extends sequential consistency, specified by condition (i). Intuitively, linearizability may be interpreted as follows: each operation appears to take effect instantaneously, at some point in time between its invocation and its termination; and the order of operations respects their "real time" order.

Linearizability thus appears to be more intuitive that sequential consistency, since it preserves the order of (non overlapping) operations. In addition, contrary to sequential consistency, it has a compositional property (a combination of separate linearizable implementations of objects is linearizable).

When dealing with replicated data, the above requirements are transposed as follows. The requirement corresponding to sequential consistency is *one-copy serializability* (1SR). This concept, introduced in database systems [Bernstein et al. 1987], captures two notions: (i) replication is invisible to the clients, thus maintaining the illusion that there exists a single (virtual) copy of a replicated object; and (ii) the operations performed on these virtual copies (actually implemented by operations on replicas) are sequentially consistent. Linearizability is likewise extended to replicated objects (in addition to one-copy serializability, the order in which the operations appear to be executed is compatible with the global ordering of operations).

The above criteria define *strong consistency*, in the sense that they maintain the illusion of a single object.

In *weak consistency*, by contrast, individual replicas are considered, and they may diverge, with some specified restrictions. Conflicts appear if these restrictions run the risk of been violated. Conflicts must be resolved (e.g., by preventing or by delaying an operation, etc.). The usual weak consistency condition is *eventual convergence*, also called *uniformity*: if no operations are performed after a certain time, all replicas must eventually converge to be identical.

Consider a service implemented by a set of replicated servers, following either of the patterns described in 11.3.1. Then a sufficient condition for linearizability is that (i) if a request is delivered to one correct replica, it must be delivered to all correct replicas; and (ii) if two requests are delivered to two correct replicas, they must be executed in the same order on both. Two schemes satisfying this condition are described in the next two subsections.

11.6.2 Primary-Backup Server Protocol

In the primary-backup scheme, all requests are sent to the primary, whose identity is known to the clients. In normal operation (i.e., if the primary is up), a request is processed as follows (Figure 11.8):

1. The primary executes the request, yielding a reply r and a new state S.

2. The primary multicasts r and S to all the backups. Each correct backup changes its state to S, stores r, and replies to the primary with an ACK message.

3. The primary waits for all the ACKs issued by the backups that it knows to be correct. It then sends the reply to the client, and waits for the next request.

Each request, and the corresponding reply, is uniquely identified by the client's identity and a unique sequence number for that client.

This scheme satisfies the linearizability condition, since a total ordering of the requests is imposed by the primary, and this order is followed by the backups[14].

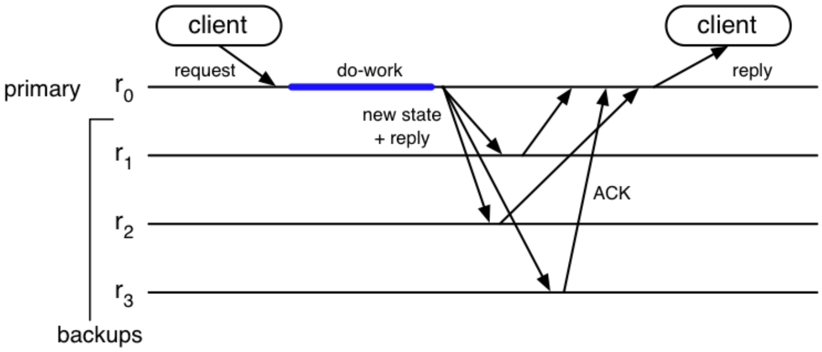

Figure 11.8. Primary-backup replication (adapted from [Guerraoui and Schiper 1997]

We now examine the case of failure. If a backup fails, there is nothing to be done, except that the primary needs to be informed of the failure (this is imposed by step 3 above: the primary does not wait for an ACK from a failed backup).

If the primary fails, a new primary must be selected among the backups, and its identity should be make known to both the clients and the remaining backups. The simplest arrangement is to specify a fixed (circular) ordering among all replicas. The new primary is the first correct replica that follows the failed primary in this ordering, which is known to both the servers and the clients. A client detects the failure of the primary by means of a timeout.

The key point of the recovery protocol is to ensure that the new primary can start operating with no request loss or duplication. The action to perform depends on the precise stage at which the failure occurred.

1. If the primary failed before multicasting the reply and new state of a request, no backup is aware of the request. Eventually, the client will time out, and resend the request to the new primary.

2. If the primary failed after the multicast, but before sending the reply to the client, the new primary will have updated its state; after the view change, it will be aware of its new role and it will send the stored reply to the client.

3. If the primary failed after having sent the reply, the client will receive a duplicated reply from the new primary. It will detect the duplication through the unique identification, and ignore the redundant reply.

[14]The backups do not actually *execute* the requests, but their state evolves as if they did.

A virtually synchronous group membership protocol answers all the demands of the above protocol. The primary uses reliable, view synchronous multicast to send the request and new state to the backups. This guarantees that there is no intermediate situation between the cases 1 and 2 of the recovery protocol (i.e., all the correct backups get the message, or none). A view change is triggered by the failure of a replica (including the primary). Thus the primary knows which backups it should expect an ACK from, while each backup knows the new primary after a failure of the current one.

When a failed replica is reinserted after repair, it triggers a view change and takes the role of a backup. Before it can actually act in that role (i.e., be able to become a primary), it needs to update its state, by querying another correct backup. In the meantime, any incoming messages from the primary should be batched for further processing.

The primary-backup replication scheme resists to $n-1$ failures if there are n replicas (primary included). If $n = 2$ (single backup), a frequent case, the protocol is simplified: there is no need to wait for an ACK from the backup, and the primary sends the reply to both the client and the backup as soon as the work is done. This case was described in [Alsberg and Day 1976].

An early application of primary-backup replication is the process pair mechanism [Bartlett 1981] used in the Tandem NonStop system, a highly available computer system [Bartlett and Spainhower 2004]. A (logical) process is implemented by a pair of processes (the primary and the backup), running on different processors. A request directed to the logical process goes through a redirection table, which sends it to the process that is the current primary. After completion of a request, the primary commands the backup to checkpoint the request and the process state. When the primary (or its processor) fails, the backup process takes over, and the redirection table is updated. When the failed primary recovers, it becomes the backup.

11.6.3 Active Replication

Active server replication follows the replicated state machine approach described in 11.3.1. A client sends its request to all the replicas, which have the same status. Each (correct) replica does the required work and sends the reply to the client. The client accepts the first reply that it gets, and discards the others, which are redundant. The client is not aware of any failures as long as at least one replica is up; thus, like primary-backup, active replication with n servers resists to $n-1$ failures.

As noted in 11.3.1, the requests need to be processed in the same order on all replicas, if the execution of a request modifies the state of the server. Therefore the client needs to use totally ordered (atomic) multicast to send its requests.

When a failed replica is reinserted after repair, it needs to update its state. To do so, it atomically multicasts a query for state to the group of replicas (including itself), and uses the first reply to restore its state. Atomic multicast ensures total order between requests for state and client requests. Thus all queried replicas reply with the same value of the state, and the reinserted replica, R, does not lose or duplicate requests, by using the following procedure. Let t be the instant at which R receives its own query for state. Client requests received by R before t are discarded; requests received between t and the first receipt of the state are batched for further processing.

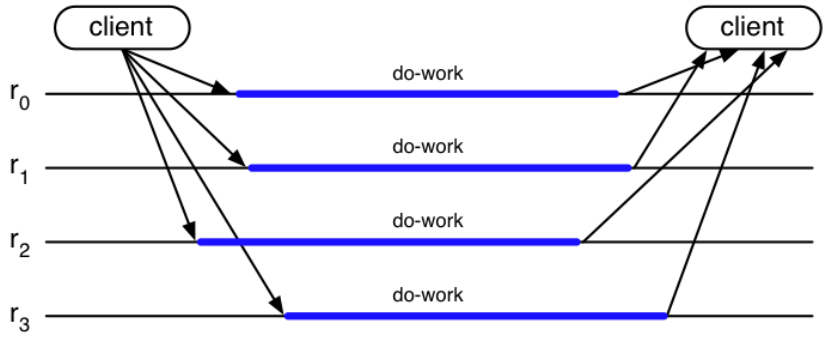

Figure 11.9. Active replication (adapted from [Guerraoui and Schiper 1997]

We now compare the primary-backup and active replication approaches. The main points to note are the following.

- Active replication consumes more resources, since the servers are dedicated to the processing of the requests. With the primary-backup scheme, backups may be used for low-priority jobs.
- Active replication has no additional latency in the case of failures, since there is no recovery protocol. In contrast, the view synchronous failure detection mechanism used in the primary-backup scheme may suffer from false failure detections, which add latency.
- Active replication is transparent for the clients, as long as one server is up. In contrast, clients need to detect the failure of the primary.
- Active replication, being based on the replicated state machine model, implies that the replicated servers behave in a deterministic way. There is no such constraint for the primary-backup scheme.
- Primary-backup uses view-synchronous group membership, while active replication uses totally ordered broadcast. In an asynchronous system with fail-stop failures, both communication protocols are equivalent to consensus. Thus the algorithmic difficulty is the same for both replication schemes.

The standard server replication scheme is primary-backup. Active replication is used in critical environments, when low latency and minimal client involvement are required.

11.6.4 Byzantine Fault Tolerance for Replicated Services

In the Byzantine failure mode, the failed component may have an arbitrary behavior. There are two main reasons to consider Byzantine failures: theoretical (this is the most general failure mode); and practical (the fail-stop hypothesis is not always verified; Byzantine failures cover the case of malicious attacks; and some critical applications require the highest possible degree of fault tolerance).

Early research on Byzantine failures has considered the problem of reliable broadcast. The main results are the following (f is the maximum number of faulty processes).

- With *synchronous* communication [Lamport et al. 1982], reliable broadcast is possible if the number of processes is at least $3f + 1$. The algorithm requires $f + 1$ rounds, and its cost (in terms of number of messages) is exponential in the number of processes. More generally, consensus can be achieved with the same degree of redundancy.
- With *asynchronous* communication [Bracha and Toueg 1985], a weak form of reliable broadcast can be achieved if the number of processes is at least $3f+1$: if the sender is correct, all correct processes deliver the value that was sent; if the sender is faulty, either all correct processes deliver the same (unspecified) value, or the algorithm does not terminate, and no correct process delivers any value.

In both cases the minimum redundancy degree is $3f+1$. In synchronous systems, this factor may be reduced to $2f + 1$ if messages can be authentified.

For a long period, research on Byzantine failures did not have much practical impact, since the algorithms were considered too expensive. The situation has changed in recent years, and several practical fault tolerance protocols dealing with Byzantine failures have been proposed. Here is a brief summary of these efforts.

[Castro and Liskov 2002] introduced Practical Byzantine Fault Tolerance (PBFT), a form of state machine replication (11.3.1) using an extension of the Paxos consensus protocol to achieve agreement on a total order for requests among all non-faulty replicas. In order to tolerate f failures, the protocol uses $3f + 1$ replicas. We give an outline of the protocol below; see the references fo a detailed description.

Like in classic Paxos (11.3.3), the protocol goes through a sequence of views. In each view, a single primary starts an agreement protocol; if the primary fails, a new view is created, with a new primary. Safety is achieved by the agreement protocol, while liveness depends on view change. All messages are authentified to prevent corruption, replay and spoofing (which could be possible under the Byzantine fault assumption).

The agreement protocol has three phases: pre-prepare, prepare, and commit. The role of the first two phases is to totally order the requests within a view. The role of the last two phases is to ensure that requests that commit (i.e., are executed and provide a result) are totally ordered across views.

In the pre-prepare phase, the primary proposes a sequence number n for a request and multicasts it to the backups in a PRE-PREPARE message. If a backup accepts this message (based on cryptographic check and uniqueness of n within the current view), it enters the prepare phase by multicasting a PREPARE message, still including n, to all replicas (including the primary). If a replica has received $2f$ PREPAREs matching the PRE-PREPARE for a request r from different backups, it marks the request as *prepared*. The following predicate is true: no two non-faulty replicas can have *prepared* requests with the same n and different contents[15] (meaning different encrypted digests).

[15]The proof of this property goes as follows: from the definition of *prepared*, at least $f + 1$ non faulty replicas must have sent a pre-prepare or prepare message for request r. If two correct replicas have prepared requests with different contents, at least one of these senders has sent two conflicting prepares or pre-prepares; but this is not possible, since these replicas are non faulty.

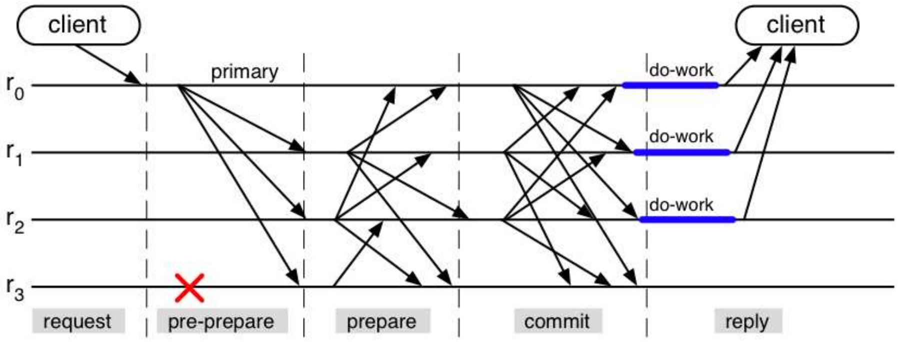

Figure 11.10. The PBFT protocol, normal case (adapted from [Castro and Liskov 1999])

When it has a *prepared* request, a replica multicasts a COMMIT message to the other replicas, thus starting the commit phase. A replica accepts this message if it is properly signed and the view in the message matches the view known as current by the replica. The commit phase ensures that the request is prepared at $f+1$ or more replicas. The key point here is that this condition can be checked by a *local* test at some replica[16] (see the original paper for a proof of this property).

After it has committed a request, a replica executes the work specified in the request and sends the result to the client. The client waits for $f+1$ matching replies. Since at most f replicas can be faulty, the result is correct.

A view change protocol detects the failure of the primary (by a timeout mechanism, which implies partial synchrony), and chooses a new primary. The new primary determines the status of pending requests and resumes normal operation.

This work has shown that Byzantine fault tolerance can be achieved at acceptable cost, and has stimulated further research with the goal of improving the performance of the protocol. Two main paths have been followed: improving throughput, by increasing the number of replicas and using quorums [Cowling et al. 2006, Abd-El-Malek et al. 2005]; improving latency, by reducing the number of phases through speculation (optimistic execution) [Kotla et al. 2007]. A generic, modular approach, which aims at combining the advantages of previous protocols, is presented in [Guerraoui et al. 2008].

11.7 Data Replication

Data replication is motivated by two concerns.

- *Availability.* Maintaining several copies of data on different systems increases the probability of having at least one available copy in the face of processor, system, or network failures.

- *Performance.* Replication allows parallel access to the systems hosting the replicas, thus increasing the global throughput for data access. Distributing replicas over a

[16]The local test verifies the following predicate: the replica has a *prepared* request, and has accepted $2f+1$ matching commits (possibly including its own) from different replicas.

network allows a client to access a replica close to its location, thus reducing latency. Both factors favor scalability.

In this section, we use the term *object* to designate the unit of replication, as chosen by the designer of the replication system. An object may be defined at the physical level (e.g., a disk block), or at the logical level (e.g., a file, or an item in a database). Unless otherwise specified, we assume that replication is based on logical objects.

The counterpart (and main challenge) of data replication is that the replicas of an object need to be kept *consistent* (as discussed in 11.6.1). Maintaining consistency has an impact on both performance and availability. Designing a data replication system thus involves a trade-off between these three properties.

The issue of data replication has been considered in two different contexts: distributed systems and databases. The main motivation of replication is availability for distributed systems, and performance for databases. The two communities have traditionally taken different approaches: database tend to closely integrate replication with data access protocols, while distributed systems use generic tools such as group communication. In recent years, efforts have been made towards a unified approach to data replication. [Wiesmann et al. 2000b] analyze the solutions developed in the two communities and compare them using a common abstract framework. This trend towards convergence has led to the emergence of a middleware-based approach to database replication, in which replication is controlled by a middleware layer located between the clients and the database replicas, and separate from the data access protocols. Separating these two concerns simplifies development and maintenance, and allows using techniques developed for distributed systems, such as group communication protocols. [Cecchet et al. 2008] examine the state of the art and the main challenges of middleware-based database replication.

In an often cited paper, [Gray et al. 1996] discuss the trade-off between consistency and performance in replicated databases. They distinguish between two approaches:

- *Eager replication*, also called *pessimistic* approach: keeping all replicas synchronized, by updating the replicas inside one atomic transaction (i.e., no access is permitted until all replicas are updated).

- *Lazy replication*, also called *optimistic* approach: performing update at one replica, and thereafter asynchronously propagating the changes to the other replicas.

Hybrid methods, combining eager and lazy replication, have also been proposed.

Eager replication maintains strong consistency (in the sense defined in 11.6.1), but it delays data access, and does not scale. In practice, its use is limited to sites located on a local area network, under moderate load. Lazy replication scales well, and may be used with unreliable communication networks. Lazy replication only ensures weak consistency, in the sense that a read may return an out of date value; in addition, an update operation runs the risk of being aborted in order to preserve consistency.

In both eager and lazy replication, a read operation may usually be performed at any site holding a replica. Regarding updates, a distinction may be made between two approaches, which correspond to the two patterns identified in 11.3.1:

- *Update primary (or single-master)*. For each object, one replica is designated as the primary copy, or master. All updates are made on the master, which is responsible for propagating them to the other replicas.

- *Update anywhere (or multi-master)*. An update operation may originate at any replica. Potential conflicts must be resolved by an agreement protocol between the involved replicas.

This distinction is orthogonal to that between eager and lazy replication; thus both eager and lazy replication may be implemented using either "update primary" or "update anywhere".

In this section, we present an overview of data replication techniques, illustrated by examples from both databases and distributed systems. Section 11.7.1 deals with strongly consistent replication. Section 11.7.2 examines weakly consistent replication. Section 11.7.3 presents a brief conclusion on data replication.

11.7.1 Strongly Consistent (Eager) Replication

Eager replication preserves a strong form of consistency, usually one-copy serializability, as defined in 11.6.1. Another consistency criterion (snapshot isolation), weaker than 1SR, is discussed later in this section.

[Wiesmann et al. 2000a] propose a classification of eager replication techniques for databases using transactions to access data. In addition to the distinction between primary copy and update everywhere, they identify two other criteria: (i) linear or constant interaction, according to whether the updates are propagated operation by operation, or in a grouped action at the end of a transaction; and (ii) voting or non voting, according to whether the different replicas execute or not a coordination phase at the end of a transaction. They conclude that linear interaction suffers from excessive overhead (due to the large number of update propagation messages), and that constant interaction should be preferred. Voting is needed if the order in which operations are performed on the replicas is non-deterministic.

We now examine the main techniques used to ensure eager replication.

Eager Update Anywhere

The basic protocol for eager replication is *read one-write all* (ROWA). An update operation on an object is performed on all the nodes holding a replica of the object; a read operation is only done on the local copy (or on the closest node holding a replica).

The drawback of this technique is that a failure of a node blocks update operations on the objects that have a replica on that node. Two solutions have been proposed to avoid this problem.

- Update only the available replicas, i.e., those located on non-failed, accessible nodes (*read one-write all available*, or ROWAA [Bernstein et al. 1987]). When a failed node recovers, it needs to update the replicas that it holds before being able to process requests.

- Use a quorum-based protocol [Gifford 1979, Thomas 1979], which requires that a minimum number of sites (a read or write *quorum*) be available for both read and write operations. The basic protocol requires that $r + w > N$, where r is the read quorum, w is the write quorum, and N the number of replicas. This guarantees that a read quorum and a write quorum always intersect; therefore, if the most recently updated copy is selected for a read, it is guaranteed to be up to date. The additional condition $w > N/2$ only allows an update if a majority of sites are available; this prevents divergence of replicas in case of a network partition. The protocol may be refined by assigning different weights to the replicas (N is now the sum of the weights).

If transactions are supported, a 2PC protocol (9.4) is needed if transactions contain writes, and 1SR is ensured by a concurrency control algorithm.

[Jiménez-Peris et al. 2003] compare ROWAA and quorum-based protocols. They conclude that ROWAA is the best solution in most practical situations, specially in terms of availability. Quorums may be of interest in the case of extreme loads.

For both methods, scalability is limited (a few tens of nodes), specially if the update rate is high. For database systems using locks to ensure concurrency control, a strong limiting factor for scalability if the probability of deadlocks, which may in some cases grow as N^3, N being the number of replicas [Gray et al. 1996].

Several approaches, often used in combination, have been proposed to overcome the limitations of ROWAA.

- Using group communication protocols to propagate update to replicas, with order guarantees, in order to reduce the probability of conflicts, and to allow increased concurrency.

- Replacing the 1SR condition by a weaker one, *snapshot isolation* (SI), see 9.2.3. Within a transaction, the first write creates a new version of the updated object[17]. Subsequent reads and writes access this version, while reads not preceded by a write return the last committed version of the object being read. Concurrent writes on the same object by two transactions are not allowed (one of the transactions must abort). Thus reads are neither blocking nor being blocked, and are therefore decoupled from writes, which increases concurrency.

- Taking advantage of cluster architectures.

We illustrate these approaches with a brief review of three examples.

1) Database State Machine. The database state machine [Pedone et al. 2003] is the transposition of the replicated state machine model (11.3.1) to databases. It uses atomic multicast to order updates. The entire database is replicated on several sites. A transaction is processed at a single site (e.g., the nearest). The identity of the objects that are read and written by the transaction (*readset* and *writeset*, respectively) is collected, as well as the values of the updates. When the transaction issues a commit command, it is immediately

[17]This is similar to the copy-on-write technique used in shared virtual memory systems.

committed if it is read only. If the transaction contains writes, the writesets, readsets and updates are atomically multicast to all sites holding a replica of the database. When a site receives this information for a transaction T, it certifies T against all transactions that conflict with T (as known to the the site). All sites execute the same certification protocol (based on concurrency control rules) and get the transaction data in the same order. Therefore, all sites will make the same decision for T (commit or abort). If the transaction commits, its updates are performed at all sites (this is called the deferred update technique). The certification test described in [Pedone et al. 2003] guarantees 1SR. The database state machine approach avoids distributed locking, which improves scalability.

The database state machine is an abstract model, which has only been evaluated by simulation. It may be considered as a basis for the development of actual protocols (for instance, the Postgres-R(SI) protocol described below is based on the same principles, with a relaxed consistency condition).

2) Postgres-R(SI): Group Communication and Snapshot Isolation. Postgres-R(SI) [Wu and Kemme 2005] is a system implementing SI, and based on group communication. A transaction can be submitted at any replica. It executes on that replica, and collects all its write operations in a writeset. When the transaction commits, it multicasts its writesets to all replicas, using uniform atomic (totally ordered) multicast. All replicas execute the operations described in the writesets they receive, in the order the writesets were delivered, which is the same for all replicas. This protocol uses less messages than ROWAA, and allows for increased concurrency, since a transaction may commit locally, without running a 2PC protocol. We do not describe the concurrency control algorithm.

Site failures are detected by the group communication protocol. The group continues working with the available sites. When a site restarts after a failure, it must run a recovery protocol to update its state, by applying the writesets that it missed (these are recovered from a log on a correct site). During this recovery process, the writesets generated by current transactions are buffered for further execution.

3) C-JDBC: Cluster-based Replicated Database. Clusters of workstations have been developed as an alternative to high-end servers. In addition to being easily extensible at a moderate cost, they allow a potentially high degree of parallelism, together with high performance communication. One proposal to exploit these features for data replication at the middleware level is Clustered JDBC, or C-JDBC [Cecchet et al. 2004]. C-JDBC is a front-end to one or several totally or partially replicated databases hosted on a cluster and accessible through the Java Database Connectivity (JDBC) API. It presents a single database view (a virtual database) to the client applications.

C-JDBC is a software implementation of the concept of RAIDb (Redundant Array of Inexpensive Databases), which is a counterpart of the existing RAIDs (Redundant Array of Inexpensive Disks). Like RAIDs, RAIDbs come in different levels, corresponding to various degrees of replication and error checking.

C-JDBC is composed of a JDBC driver, which exports the interface visible to the client applications, and a controller, which handles load balancing and fault tolerance, and acts as a proxy between the C-JDBC driver and the database back-ends.

Operations are synchronous, i.e., the controller waits until it has a response from all back-ends involved in an operation before it sends a response to a client. To reduce latency for the client, C-JDBC also implements early response, i.e., the controller returns the response as soon as a preset number of back-ends (e.g., one, or a majority) has executed an operation. In that case, the communication protocol ensures that the operations are executed in the same order at all involved back-ends.

C-JDBC does not use a 2PC protocol. If a node fails during the execution of an operation, it is disabled. After repair, a log-based recovery protocol restores an up to date state.

The controller is a single point of failure in C-JDBC. To improve availability, the controller may be replicated[18]. To ensure mutual consistency, the replicated controllers use totally ordered group communication to synchronize write requests and transaction demarcation commands.

A system inspired by C-JDBC is Ganymed [Plattner and Alonso 2004], which implements snapshot isolation in order to reduce the probability of conflicts. Ganymed uses a primary copy approach, and the controller ensures that updates are done in the same order at the backup replicas.

In conclusion, update anywhere for eager replication may be summarized as follows. ROWAA is a simple protocol for non-transactional operations. If transactions are used, the traditional technique based on ROWAA is hampered by the deadlock risks of distributed locking and by the cost of distributed commitment. Alternative approaches for transactions involve the use of atomic multicast to propagate updates, and a weakening of the one-copy serializability condition.

Eager Primary Copy

In the primary copy replication technique, one particular site holding a replica of an object is designated as the primary for that object (note that different subsets of data may have different primary sites); the other sites holding replicas are backups. A transaction which updates an object is executed at the primary for that object. A the end of the transaction, the updates are propagated to the backups in a single operation, which groups the changes in FIFO order. If the technique is non-voting, the primary commits the transaction. If it uses voting (in practice, applying a 2PC protocol), the primary and the backups must wait till the conclusion of the vote. Some remarks are in order.

- If there are several primaries, the situation is close to that of an update anywhere technique[19] described above. In the non-voting scheme, the communication protocol must guarantee total order for the updates at the backup sites. In a voting scheme, no order guarantee is needed, but the transaction runs the risk of being aborted; a weak order guarantee (such as FIFO) reduces the risk of abort.

[18] This is called "horizontal" replication, in contrast with "vertical" replication, which consists in building a tree of controllers to support a large number of back-ends.

[19] For that reason, some primary copy systems disallow transactions that update objects with different primary sites.

- In a non-voting protocol, a read operation is only guaranteed to deliver an up to date result if it is performed at the primary (or if the primary does not commit before all backups have been updated). Otherwise, the situation is the same as in lazy replication (11.7.2).

- If a primary fails, the recovery process is different for voting and non-voting techniques. With the voting technique, a backup is always ready to become primary (this is called hot standby). With the non-voting technique, the backup may have pending updates, and needs to install them before becoming primary (this is called cold standby).

In conclusion, primary copy for eager replication is essentially used in two modes: (i) backup copies are used for reads, in which case a voting protocol must be used; or (ii) backups are used for recovery only, in which case all operations are done on the primary.

An early example of this latter use is Tandem's Remote Data Facility [Lyon 1990], in which there is a single backup, and updates are immediately propagated, thus ensuring hot standby.

11.7.2 Loosely Consistent (Lazy) Replication

Lazy replication trades consistency for efficiency; fault tolerance is less of a priority than in eager replication. After recovery from a failure, some updates may have been lost. Lazy replication may be implemented using the primary copy or the update anywhere techniques. In both cases, the main consistency criterion is eventual convergence (as defined in 11.6.1).

Lazy Primary Copy

With the lazy primary copy (or single master) technique, all updates are done on a designated site, the primary[20]. All operations commit immediately. Updates are propagated asynchronously to the backup sites, using either a push or a pull method. Read operations may be done either on the primary or on the backups; in the latter case, an out of date value may be returned, which is acceptable in many applications. Some systems allow a given degree of freshness to be specified for reads, e.g., in terms of maximum age.

Since all updates are done on a single site, potential conflicts between concurrent updates are easily detected and may be avoided by delaying or aborting some operations.

Because of its simplicity, lazy primary copy is the most commonly used technique in commercial database systems. Its main drawback is that the primary site is a single point of failure. If the primary fails, a backup is selected as the new primary, but some updates may have been lost.

One technique that attempts to avoid the loss of updates is to keep a log of updates in stable (failure-resistant) storage. This technique is used by the Slony-I system [Slony-I], which supports several back-ups, possibly organized as a cascade. This technique attempts to approximate hot standby, while keeping the benefits of primary copy update.

[20] In some systems, different objects may have different primary sites.

Lazy Update Anywhere

Lazy update anywhere (or multi-master) techniques are intended to be more efficient than those based on primary copy, by increasing concurrency. The counterpart is that conflicts may occur between concurrent updates at different sites; such conflicts must be detected and resolved. For these systems, fault tolerance is not the primary concern.

[Saito and Shapiro 2005] is an extensive survey of lazy (optimistic) update anywhere replication approaches, mostly for non-database application. They distinguish between state-transfer and operation-transfer systems. In the former, the replicas of an object are updated by copying their contents. In the latter, the operations are propagated to the replicas and executed there.

For state-transfer systems, Thomas's write rule [Thomas 1979] ensures uniformity. Recall (9.2.3) that this rule is based on timestamps; a replica of an object is updated when it detects (e.g., by periodic inspection) a peer holding a more recent copy of the object. Conflicts between concurrent updates are resolved by the "last writer wins" policy.

For operation transfer systems, the situation is analogous to that described in 11.7.1: a sufficient condition for consistency is that the updates be applied at each replica in the same order. This order may be imposed by the sender, through atomic broadcast, or determined by the receivers, using time-stamp vectors. As above, one can use the semantics of the application to loosen the constraint on the order of updates (for example, non-interfering updates may be applied in any order).

11.7.3 Conclusion on Data Replication

Several issues need to be considered for data replication: implemented by a middleware layer or integrated in the application, eager vs lazy update, what consistency guarantees?

Eager replication, which gives strong consistency guarantees, is gaining favor as new designs allow improved performance, specially on clusters. Lazy replication is mandatory in systems that are highly dynamic or have network reliability or connectivity problems, such as MANETs (mobile ad-hoc networks). Such systems also rely on probabilistic techniques (gossip protocols), such as described in 11.4.4.

In practice, essentially for performance reasons, many commercial database systems use primary-based, lazy update replication, integrated within the database kernel. Availability relies on hot standby solutions.

Solutions based on middleware-based systems are investigated. Current efforts are based on group protocols, and tend to favor the eager, update anywhere approach, with snapshot isolation. There is still a gap between current research and actual practice (see [Cecchet et al. 2008]).

11.8 Partial Availability

This section still unavailable.

11.9 Case Studies

This section still unavailable.

11.10 Historical Note

Fault tolerance has always been a primary concern in computing systems. In the early days, attention was mainly focused on unreliable hardware. Error detecting and correcting codes were used to deal with unreliable memory and communication links. Backup techniques (saving drum or disk contents on magnetic tapes) were used for data preservation, with the risk of data loss in the intervals between backups. Later on, disk mirroring techniques improved the situation, but their use was limited by their high relative cost. To deal with processor failures, Triple Modular Redundancy was introduced for critical applications. Software reliability only started receiving attention by the end of the 1960s.

The 1970s are a period of fast progress for computing systems availability. The first *International Symposium on Fault Tolerant Computing* (FTCS) takes place in 1971; it is still mostly devoted to hardware aspects, but software based techniques are beginning to develop. An early attempt at a purely software approach to fault tolerance is the concept of recovery block [Randell 1975], based on redundant programming of critical routines. The design of the SIFT aircraft control system [Wensley et al. 1976] is a large scale effort, mainly using software methods, to build a reliable critical application.

The Tandem highly available systems (1976) mark a breakthrough. While previous fault tolerance techniques were essentially application specific, the Tandem NonStop system [Bartlett 1981] introduces a generic approach combining hardware and software replication (most notably, process pairs). However, process checkpointing is a source of overhead. The Stratus systems (1980) eliminates overhead by a purely hardware approach, at the expense of quadruple redundancy (2 duplexed pairs of processors, the processors of each pair working in lockstep and performing a continuous consistency check).

One of the first available server systems based on the primary-backup approach (with a single backup) is described in [Alsberg and Day 1976]. The replicated state machine paradigm is introduced in [Lamport 1978a], and further refined in [Schneider 1990]. The atomic commitment problem, and its first solution (2PC), are presented in [Gray 1978]. One of the first impossibility results (widely known as "the Generals' paradox" from [Gray 1978]), related to unreliable communication, is published in [Akkoyunlu et al. 1975]. An influential paper on data availability is [Lampson and Sturgis 1979], which introduces the notion of stable storage. Algorithms for database replication are studied, and the concepts of quorum and majority voting are introduced [Gifford 1979, Thomas 1979].

The 1980's and early 90's are marked by a sustained effort to ground the design and construction of fault tolerant systems on a sound fundamental base. Computing systems become distributed; the first *Symposium on Reliable Distributed Systems* (SRDS) takes place in 1981. As a consequence, communication aspects take an increasing importance. Group communication is identified as an essential ingredient of fault tolerant distributed computing, and the central role of agreement protocols (consensus and atomic commitment) is recognized:

- Paxos [Lamport 1998], an efficient consensus algorithm, is published in 1989, but its significance will only be fully recognized much later.
- Unreliable failure detectors are introduced in [Chandra and Toueg 1991]; this paper also gives a consensus-based implementation of atomic broadcast.
- The atomic commitment problem is further explored; a non-blocking solution is proposed in [Skeen 1981]; further advances are described in [Babaoğlu and Toueg 1993]. Advances in fault-tolerant database systems are described in [Bernstein et al. 1987].

The notion of virtual synchrony, which associates group membership and communication, appears in [Birman and Joseph 1987].

Byzantine agreement (a problem that was already identified during the design of the SIFT system) is explored, and its impossibility for $3f$ processes with f faults is proven [Lamport et al. 1982]. A number of other impossibility results, including FLP [Fischer et al. 1985] are discovered; a review of these results is in [Lynch 1989].

The first mention of gossip-based communication appears in [Demers et al. 1987], in a replicated database context, but the idea will only be put to wider practical use a decade later.

The main advance in hardware-based availability in this period is the invention of the RAIDs [Patterson et al. 1988]. As hardware becomes more reliable, the increasing role of software and administration faults is recognized [Gray 1986].

[Cristian 1991] describes the state of the art in fault-tolerant distributed systems at the end of this period.

The mid and late 1990's see the development of distributed systems: cluster computing, distributed transactions, database replication, fault-tolerant middleware. Group communication is now well understood ([Hadzilacos and Toueg 1993] present a survey of this area, which clarifies the concepts and terminology), and used in practical systems. The interplay between theory and practice in the areas of group communication and replication, and its evolution over time, is analyzed in [Schiper 2003, Schiper 2006b].

The 2000's are dominated by two main trends: the rise of Internet computing and services, and the increasing use of mobile devices and ad hoc networking.

An analysis of Internet services failures [Oppenheimer et al. 2003] confirms the trends identified in [Gray 1986]: it shows the decreasing importance of hardware faults, and the dominating role of management related failure causes, such as wrong configuration. A consequence of large scale is the fact that all elements of a system cannot be expect to be fully operational at any point in time. Partial availability is bound to be the rule. The importance of fast failure detection and recovery is emphasized, as illustrated, for example, by project ROC [ROC 2005].

As the size and dynamism of computing systems increases, there is a need for new communication paradigms. Probabilistic methods, such as gossip based broadcast, appear as a promising path in large scale systems communication (see [Kermarrec and van Steen 2007]).

Concerning the interplay between theory and practice, the treatment of Byzantine failures receives an increasing attention. Byzantine failures are shown to be tractable in practice with an acceptable cost, as illustrated by the experiments described in 11.6.4

References

[Abd-El-Malek et al. 2005] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. (2005). Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*, pages 59–74. *Operating Systems Review*, 39(5), December 2005.

[Aguilera et al. 2000] Aguilera, M. K., Chen, W., and Toueg, S. (2000). Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125.

[Akkoyunlu et al. 1975] Akkoyunlu, E. A., Ekanadham, K., and Huber, R. V. (1975). Some constraints and tradeoffs in the design of network communications. *SIGOPS Operating Systems Review*, 9(5):67–74.

[Alsberg and Day 1976] Alsberg, P. A. and Day, J. D. (1976). A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'76)*, pages 627–644, San Francisco, California, USA.

[Appia] Appia. Appia comunication framework. http://appia.di.fc.ul.pt.

[Aspnes 2003] Aspnes, J. (2003). Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2-3):165–175.

[Avižienis et al. 2004] Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33.

[Babaoğlu and Toueg 1993] Babaoğlu, Ö. and Toueg, S. (1993). Non-Blocking Atomic Commitment. In Mullender, S., editor, *Distributed Systems*, pages 147–168. Addison-Wesley.

[Ban 1998] Ban, B. (1998). Design and Implementation of a Reliable Group Communication Toolkit for Java. Technical Report, Dept. of Computer Science, Cornell University, September 1998. http://www.jgroups.org/.

[Bartlett 1981] Bartlett, J. F. (1981). A NonStop Kernel. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 22–29, New York, NY, USA. ACM.

[Bartlett and Spainhower 2004] Bartlett, W. and Spainhower, L. (2004). Commercial Fault tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96.

[Bernstein et al. 1987] Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. 370 pp.

[Birman and Joseph 1987] Birman, K. P. and Joseph, T. A. (1987). Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP'87)*, pages 123–138.

[Blahut 2003] Blahut, R. (2003). *Algebraic Codes for Data Transmission*. Cambridge University Press. 482 pp.

[Bracha and Toueg 1985] Bracha, G. and Toueg, S. (1985). Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840.

[Candea et al. 2004] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G., and Fox, A. (2004). Microreboot – A Technique for Cheap Recovery. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI'04)*, pages 31–44, San Francisco, CA, USA.

[Castro and Liskov 1999] Castro, M. and Liskov, B. (1999). Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, Louisiana.

[Castro and Liskov 2002] Castro, M. and Liskov, B. (2002). Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461. Based on work published in *Usenix OSDI'99* (173–186) and *OSDI'00* (273–288).

[Cecchet et al. 2008] Cecchet, E., Ailamaki, A., and Candea, G. (2008). Middleware-based Database Replication: The Gaps between Theory and Practice. In *SIGMOD'08: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, June 9–12 2008, Vancouver, Canada.

[Cecchet et al. 2004] Cecchet, E., Marguerite, J., and Zwaenepoel, W. (2004). C-JDBC: Flexible Database Clustering Middleware. In *Proc. USENIX Annual Technical Conference, Freenix Track*, Boston, MA, USA.

[Chandra et al. 2007] Chandra, T. D., Griesemer, R., and Redstone, J. (2007). Paxos made live: an engineering perspective. In *Proceedings of the Twenty-sixth ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 398–407, New York, NY, USA. ACM.

[Chandra et al. 1996a] Chandra, T. D., Hadzilacos, V., and Toueg, S. (1996a). The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722.

[Chandra et al. 1996b] Chandra, T. D., Hadzilacos, V., Toug, S., and Charron-Bost, B. (1996b). On the Impossibility of Group Membership. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*.

[Chandra and Toueg 1991] Chandra, T. D. and Toueg, S. (1991). Unreliable failure detectors for reliable distributed systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 325–340.

[Chandra and Toueg 1996] Chandra, T. D. and Toueg, S. (1996). Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267. A preliminary version appeared in [Chandra and Toueg 1991].

[Chandy and Lamport 1985] Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75.

[Chen et al. 2002] Chen, W., Toueg, S., and Aguilera, M. K. (2002). On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580.

[Chockler et al. 2001] Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group Communication Specifications: a Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469.

[Cowling et al. 2006] Cowling, J., Myers, D., Liskov, B., Rodrigues, R., and Shrira, L. (2006). HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI'06)*, pages 177–190, Seattle, WA, USA.

[Cristian 1991] Cristian, F. (1991). Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78.

[Défago et al. 2004] Défago, X., Schiper, A., and Urbán, P. (2004). Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys*, 36(4):372–421.

[Demers et al. 1987] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing (PODC '87)*, pages 1–12, New York, NY, USA. ACM.

[Dwork et al. 1988] Dwork, C., Lynch, N., and Stockmeyer, L. (1988). Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323.

[Elnozahy et al. 2002] Elnozahy, E. N. M., Alvisi, L., Wang, Y.-M., and Johnson, D. B. (2002). A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408.

[Fischer et al. 1985] Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985). Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382.

[Gifford 1979] Gifford, D. K. (1979). Weighted voting for replicated data. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP'79)*, pages 150–162.

[Gray 1978] Gray, J. (1978). Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK. Springer-Verlag.

[Gray 1986] Gray, J. (1986). Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12.

[Gray et al. 1996] Gray, J., Helland, P., O'Neil, P., and Shasha, D. (1996). The dangers of replication and a solution. In *SIGMOD'96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182.

[Gray and Reuter 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann. 1070 pp.

[Guerraoui et al. 2008] Guerraoui, R., Quéma, V., and Vukolić, M. (2008). The next 700 BFT protocols. Technical Report LPD-2008-08, École Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences, Lausanne, Switzerland.

[Guerraoui and Schiper 1997] Guerraoui, R. and Schiper, A. (1997). Software-Based Replication for Fault Tolerance. *IEEE Computer*, 30(4):68–74.

[Guerraoui and Schiper 2001] Guerraoui, R. and Schiper, A. (2001). The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41.

[Hadzilacos and Toueg 1993] Hadzilacos, V. and Toueg, S. (1993). Fault-Tolerant Broadcasts and Related Problems. In Mullender, S., editor, *Distributed Systems*, pages 97–168. Addison-Wesley.

[Herlihy and Wing 1990] Herlihy, M. P. and Wing, J. M. (1990). Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492.

[Jiménez-Peris et al. 2003] Jiménez-Peris, R., Patiño-Martínez, M., Alonso, G., and Kemme, B. (2003). Are quorums an alternative for data replication? *ACM Transactions on Database Systems*, 28(3):257–294.

[Kermarrec and van Steen 2007] Kermarrec, A.-M. and van Steen, M., editors (2007). *Gossip-Based Computer Networking*, volume 41(5), Special Topic of *ACM Operating Systems Review*.

[Kotla et al. 2007] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. (2007). Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, pages 45–58. *Operating Systems Review*, 41(6), December 2007.

[Lamport 1978a] Lamport, L. (1978a). The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114.

[Lamport 1978b] Lamport, L. (1978b). Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–56.

[Lamport 1979] Lamport, L. (1979). How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 9(C-28):690–691.

[Lamport 1998] Lamport, L. (1998). The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169. First appeared as DEC-SRC Research Report 49, 1989.

[Lamport et al. 1982] Lamport, L., Shostak, R. E., and Pease, M. C. (1982). The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401.

[Lampson 2001] Lampson, B. (2001). The ABCDs of Paxos. In *Proc. ACM Symposium on Principles of Distributed Computing (PODC)*.

[Lampson and Sturgis 1979] Lampson, B. W. and Sturgis, H. E. (1979). Crash recovery in a distributed data storage system. Technical report, Xerox PARC (unpublished), 25 pp. Partially reproduced in: *Distributed Systems – Architecture and Implementation*, ed. Lampson, Paul, and Siegert, Lecture Notes in Computer Science 105, Springer, 1981, pp. 246–265 and pp. 357–370.

[Laprie 1985] Laprie, J.-C. (1985). Dependable Computing and Fault Tolerance: Concepts and Terminology. In *Proceedings of the 15th IEEE International Symposium on Faul-Tolerant Computing (FTCS-15)*, pages 2–11.

[Larrea et al. 2004] Larrea, M., Fernández, A., and Arévalo, S. (2004). On the Implementation of Unreliable Failure Detectors in Partially Synchronous Systems. *IEEE Transactions on Computers*, 53(7):815–828.

[Larrea et al. 2007] Larrea, M., Lafuente, A., Soraluze, I., and nas, R. C. (2007). Designing Efficient Algorithms for the Eventually Perfect Failure Detector Class. *Journal of Software*, 2(4):1–11.

[Leveson and Turner 1993] Leveson, N. and Turner, C. S. (1993). An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41.

[Lions et al. 1996] Lions, J.-L. et al. (1996). Ariane 5 Flight 501 Failure Report by the Inquiry Board. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html.

[Lynch 1989] Lynch, N. (1989). A hundred impossibility proofs for distributed computing. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 1–28, New York, NY. ACM Press.

[Lyon 1990] Lyon, J. (1990). Tandem's remote data facility. In *Proceedings of IEEE Spring CompCon*, pages 562–567.

[Mostefaoui et al. 2003] Mostefaoui, A., Mourgaya, E., and Raynal, M. (2003). Asynchronous Implementation of Failure Detectors. In *International Conference on Dependable Systems and Networks (DSN'03)*, pages 35–360, Los Alamitos, CA, USA. IEEE Computer Society.

[Oki and Liskov 1988] Oki, B. M. and Liskov, B. H. (1988). Viewstamped Replication: A New Primary Copy Method to Support. Highly-Available Distributed Systems. In *Proceedings of the Seventh annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, New York, NY, USA. ACM.

[Oppenheimer et al. 2003] Oppenheimer, D., Ganapathi, A., and Patterson, D. A. (2003). Why do Internet services fail, and what can be done about it? In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*.

[Patterson et al. 1988] Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (raid). In *SIGMOD'88: Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, New York, NY, USA. ACM.

[Pedone et al. 2003] Pedone, F., Guerraoui, R., and Schiper, A. (2003). The Database State Machine Approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98.

[Plattner and Alonso 2004] Plattner, C. and Alonso, G. (2004). Ganymed: scalable replication for transactional web applications. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware (Middleware'04)*, pages 155–174, Toronto, Canada. Springer-Verlag New York, Inc.

[Randell 1975] Randell, B. (1975). System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable software*, pages 437–449, New York, NY, USA. ACM.

[ROC 2005] ROC (2005). The Berkeley/Stanford Recovery-Oriented Computing (ROC) Project. http://roc.cs.berkeley.edu.

[Saito and Shapiro 2005] Saito, Y. and Shapiro, M. (2005). Optimistic replication. *ACM Computing Surveys*, 37(1):42–81.

[Saltzer et al. 1984] Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288.

[Schiper 2003] Schiper, A. (2003). Practical impact of group communication theory. In et al., A. S., editor, *Future Directions in Distributed Computing (FuDiCo'02)*, volume 2584 of *Lecture Notes in Computer Science*, pages 1–10. Springer-Verlag.

[Schiper 2006a] Schiper, A. (2006a). Dynamic Group Communication. *Distributed Computing*, 18(5):359–374.

[Schiper 2006b] Schiper, A. (2006b). Group communication: From practice to theory. In *SOFSEM 2006: Theory and Practice of Computer Science*, number 3831 in Lecture Notes in Computer Science, pages 117–136. Springer-Verlag.

[Schiper and Toueg 2006] Schiper, A. and Toueg, S. (2006). From Set Membership to Group Membership: A Separation of Concerns. *IEEE Transactions on Dependable and Secure Computing*, 13(2):2–12.

[Schneider 1990] Schneider, F. B. (1990). Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319.

[Schneider 1993] Schneider, F. B. (1993). The Primary-Backup Approach. In Mullender, S., editor, *Distributed Systems*, chapter 8, pages 199–216. ACM Press Books, Addison-Wesley.

[Skeen 1981] Skeen, D. (1981). Non-blocking commit protocols. In *SIGMOD'81: Proceedings of the 1981 ACM-SIGMOD International Conference on Management of Data*, pages 133–142, Orlando, FL, USA.

[Slony-I] Slony-I. Enterprise-level replication system. http://slony.info/.

[Spread] Spread. The Spread toolkit. http://www.spread.org/.

[Thomas 1979] Thomas, R. H. (1979). A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209.

[van Renesse et al. 1998] van Renesse, R., Birman, K., Hayden, M., Vaysburd, A., and Karr, D. (1998). Building adaptive systems using Ensemble. *Software–Practice and Experience*, 28(9):963–979.

[Wensley et al. 1976] Wensley, J. H., Green, M. W., Levitt, K. N., and Shostak, R. E. (1976). The design, analysis, and verification of the SIFT fault tolerant system. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, pages 458–469, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Wiesmann et al. 2000a] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000a). Database replication techniques: a three parameter classification. In *Proceedings of 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, Nürenberg, Germany. IEEE Computer Society.

[Wiesmann et al. 2000b] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., and Alonso, G. (2000b). Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan. IEEE Computer Society Technical Commitee on Distributed Processing.

[Wu and Kemme 2005] Wu, S. and Kemme, B. (2005). Postgres-R(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation. In *IEEE International Conference on Data Engineering (ICDE)*, Tokyo, Japan.