CHAPTER 8

FAULT TOLERANCE

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure: part of the system is failing while the remaining part continues to operate, and seemingly correctly. An important goal in distributed systems design is to construct the system in such a way that it can automatically recover from partial failures without seriously affecting the overall performance. In particular, whenever a failure occurs, the system should continue to operate in an acceptable way while repairs are being made. In other words, a distributed system is expected to be fault tolerant.

In this chapter, we take a closer look at techniques to achieve fault tolerance. After providing some general background, we will first look at process resilience through process groups. In this case, multiple identical processes cooperate providing the appearance of a single logical process to ensure that one or more of them can fail without a client noticing. A specifically difficult point in process groups is reaching consensus among the group members on which client-requested operation to perform.

Achieving fault tolerance and reliable communication are strongly related. Next to reliable client-server communication we pay attention to reliable group communication and notably atomic multicasting. In the latter case, a message is delivered to all nonfaulty processes in a group, or to none at all. Having atomic multicasting makes development of fault-tolerant solutions much easier.

Atomicity is a property that is important in many applications. Perhaps best known in the case of database transactions, atomicity extends to distributed transactions, which we discuss separately. In particular, we pay attention to what are known as distributed commit protocols by which a group of processes are conducted to either jointly commit their local work, or collectively abort and return to a previous system state.

Finally, we will examine how to recover from a failure. In particular, we consider when and how the state of a distributed system should be saved to allow recovery to that state later on.

8.1 Introduction to fault tolerance

Fault tolerance has been subject to much research in computer science. In this section, we start with presenting the basic concepts related to processing failures, followed by a discussion of failure models. The key technique for handling failures is redundancy, which is also discussed. For more general information on fault tolerance in distributed systems, see, for example [Jalote, 1994; Shooman, 2002] or [Koren and Krishna, 2007].

8.1.1 Basic concepts

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults. Being fault tolerant is strongly related to what are called **dependable systems**. Dependability is a term that covers a number of useful requirements for distributed systems including the following [Kopetz and Verissimo, 1993]:

- Availability
- Reliability
- Safety
- Maintainability

Availability is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.

Reliability refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability. If a system goes down on average for one, seemingly random millisecond every hour, it has an availability of more than 99.9999 percent, but is still unreliable. Similarly, a system that never crashes but is shut down for two specific weeks every August has high reliability but only 96 percent availability. The two are not the same.

Safety refers to the situation that when a system temporarily fails to operate correctly, no catastrophic event happens. For example, many process-control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous. Many examples from the past (and probably many more yet to come) show how hard it is to build safe systems.

Finally, **maintainability** refers to how easily a failed system can be repaired. A highly maintainable system may also show a high degree of availability, especially

if failures can be detected and repaired automatically. However, as we shall see later in this chapter, automatically recovering from failures is easier said than done.

Note 8.1 (More information: Traditional metrics)
We can be a bit more precise when it comes to describing availability and reliability. Formally, the availability $A(t)$ of a component in the time interval $[0, t)$ is defined as the average fraction of time that the component has been functioning correctly during that interval. The **long-term availability** A of a component is defined as $A(\infty)$.

Likewise, the reliability $R(t)$ of a component in the time interval $[0, t)$ is formally defined as the conditional probability that it has been functioning correctly during that interval given that it was functioning correctly at time $T = 0$. Following Pradhan [1996], to establish $R(t)$ we consider a system of N identical components. Let $N_0(t)$ denote the number of correctly operating components at time t and $N_1(t)$ the number of failed components. Then, clearly,

$$R(t) = \frac{N_0(t)}{N} = 1 - \frac{N_1(t)}{N} = \frac{N_0(t)}{N_0(t) + N_1(t)}$$

The rate at which components are failing can be expressed as the derivative $dN_1(t)/dt$. Dividing this by the number of correctly operating components at time t gives us the **failure rate function** $z(t)$:

$$z(t) = \frac{1}{N_0(t)} \frac{dN_1(t)}{dt}$$

From

$$\frac{dR(t)}{dt} = -\frac{1}{N} \frac{dN_1(t)}{dt}$$

it follows that

$$z(t) = \frac{1}{N_0(t)} \frac{dN_1(t)}{dt} = -\frac{N}{N_0(t)} \frac{dR(t)}{dt} = -\frac{1}{R(t)} \frac{dR(t)}{dt}$$

If we make the simplifying assumption that a component does not age (and thus essentially has no wear-out phase), its failure rate will be constant, i.e., $z(t) = z$, implying that

$$\frac{dR(t)}{dt} = -zR(t)$$

Because $R(0) = 1$, we obtain

$$R(t) = e^{-zt}$$

In other words, if we ignore aging of a component, we see that a constant failure rate leads to a reliability following an exponential distribution, having the form shown in Figure 8.1.

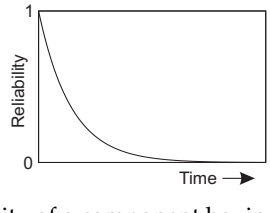

Figure 8.1: The reliability of a component having a constant failure rate.

Traditionally, fault-tolerance has been related to the following three metrics:

- **Mean Time To Failure** (***MTTF***): The average time until a component fails.
- **Mean Time To Repair** (***MTTR***): The average time needed to repair a component.
- **Mean Time Between Failures** (***MTBF***): Simply $MTTF + MTTR$.

Note that

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$$

Also, these metrics make sense only if we have an accurate notion of what a failure actually is. As we will encounter later, identifying the occurrence of a failure may actually not be so obvious.

Often, dependable systems are also required to provide a high degree of security, especially when it comes to issues such as integrity. We will discuss security in the next chapter.

A system is said to **fail** when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided. An **error** is a part of a system's state that may lead to a failure. For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver. Damaged in this context means that the receiver may incorrectly sense a bit value (e.g., reading a 1 instead of a 0), or may even be unable to detect that something has arrived.

The cause of an error is called a **fault**. Clearly, finding out what caused an error is important. For example, a wrong or bad transmission medium may easily cause packets to be damaged. In this case, it is relatively easy to remove the fault. However, transmission errors may also be caused by bad weather conditions such as in wireless networks. Changing the weather to reduce or prevent errors is a bit trickier.

As another example, a crashed program is clearly a failure, which may have happened because the program entered a branch of code containing a programming bug (i.e., a programming error). The cause of that bug is typically a programmer. In other words, the programmer is the fault of the error (programming bug), in turn leading to a failure (a crashed program).

Building dependable systems closely relates to controlling faults. As explained by Avizienis et al. [2004], a distinction can be made between preventing, tolerating, removing, and forecasting faults. For our purposes, the most important issue is **fault tolerance**, meaning that a system can provide its services even in the presence of faults. For example, by applying error-correcting codes for transmitting packets, it is possible to tolerate, to a certain extent, relatively poor transmission lines and reducing the probability that an error (a damaged packet) may lead to a failure.

Faults are generally classified as transient, intermittent, or permanent. **Transient faults** occur once and then disappear. If the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost

bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.

An **intermittent fault** occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, when the fault doctor shows up, the system works fine.

A **permanent fault** is one that continues to exist until the faulty component is replaced. Burnt-out chips, software bugs, and disk-head crashes are examples of permanent faults.

8.1.2 Failure models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with one another and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do. However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else.

Such dependency relations appear in abundance in distributed systems. A failing disk may make life difficult for a file server that is designed to provide a highly available file system. If such a file server is part of a distributed database, the proper working of the entire database may be at stake, as only part of its data may be accessible.

To get a better grasp on how serious a failure actually is, several classification schemes have been developed. One such scheme is shown in Figure 8.2, and is based on schemes described in Cristian [1991] and Hadzilacos and Toueg [1993].

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response lies outside a specified time interval
Response failure	A server's response is incorrect
Value failure	The value of the response is wrong
State-transition failure	The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 8.2: Different types of failures.

A **crash failure** occurs when a server prematurely halts, but was working cor-

rectly until it stopped. An important aspect of crash failures is that once the server has halted, nothing is heard from it anymore. A typical example of a crash failure is an operating system that comes to a grinding halt, and for which there is only one solution: reboot it. Many personal computer systems suffer from crash failures so often that people have come to expect them to be normal. Consequently, moving the reset button from the back of a cabinet to the front was done for good reason. Perhaps one day it can be moved to the back again, or even removed altogether.

An **omission failure** occurs when a server fails to respond to a request. Several things might go wrong. In the case of a **receive-omission failure**, possibly the server never got the request in the first place. Note that it may well be the case that the connection between a client and a server has been correctly established, but that there was no thread listening to incoming requests. Also, a receive-omission failure will generally not affect the current state of the server, as the server is unaware of any message sent to it.

Likewise, a **send-omission failure** happens when the server has done its work, but somehow fails in sending a response. Such a failure may happen, for example, when a send buffer overflows while the server was not prepared for such a situation. Note that, in contrast to a receive-omission failure, the server may now be in a state reflecting that it has just completed a service for the client. As a consequence, if the sending of its response fails, the server has to be prepared for the client to reissue its previous request.

Other types of omission failures not related to communication may be caused by software errors such as infinite loops or improper memory management by which the server is said to "hang."

Another class of failures is related to timing. **Timing failures** occur when the response lies outside a specified real-time interval. As we saw with isochronous data streams in Chapter 4, providing data too soon may easily cause trouble for a recipient if there is not enough buffer space to hold all the incoming data. More common, however, is that a server responds too late, in which case a *performance* failure is said to occur.

A serious type of failure is a **response failure**, by which the server's response is simply incorrect. Two kinds of response failures may happen. In the case of a value failure, a server simply provides the wrong reply to a request. For example, a search engine that systematically returns Web pages not related to any of the search terms used, has failed.

The other type of response failure is known as a **state-transition failure**. This kind of failure happens when the server reacts unexpectedly to an incoming request. For example, if a server receives a message it cannot recognize, a state-transition failure happens if no measures have been taken to handle such messages. In particular, a faulty server may incorrectly take default actions it should never have initiated.

The most serious are **arbitrary failures**, also known as **Byzantine failures**. In effect, when arbitrary failures occur, clients should be prepared for the worst. In particular, it may happen that a server is producing output it should never have

produced, but which cannot be detected as being incorrect. Byzantine failures were first analyzed by Pease et al. [1980] and Lamport et al. [1982]. We return to such failures below.

Note 8.2 (More information: Omission and commission failures)
It has become somewhat of a habit to associate the occurrence of Byzantine failures with maliciously operating processes. The term "Byzantine" refers to the Byzantine Empire, a time (330–1453) and place (the Balkans and modern Turkey) in which endless conspiracies, intrigue, and untruthfulness were alleged to be common in ruling circles.

However, it may not be possible to detect whether an act was actually benign or malicious. Is a networked computer running a poorly engineered operating system that adversely affects the performance of other computers acting maliciously? In this sense, it is better to make the following distinction, which effectively excludes judgment:

- An **omission failure** occurs when a component fails to take an action that it should have taken.
- A **commission failure** occurs when a component takes an action that it should not have taken.

This difference, introduced by Mohan et al. [1983], also illustrates that there may indeed be a thin line between dependability and security.

Many of the aforementioned cases deal with the situation that a process P no longer perceives any actions from another process Q. However, can P conclude that Q has indeed come to a halt? To answer this question, we need to make a distinction between two types of distributed systems:

- In an **asynchronous system**, no assumptions about process execution speeds or message delivery times are made. The consequence is that when process P no longer perceives any actions from Q, it cannot conclude that Q crashed. Instead, it may just be slow or its messages may have been lost.
- In a **synchronous system**, process execution speeds and message-delivery times are bounded. This also means that when Q shows no more activity when it is expected to do so, process P can rightfully conclude that Q has crashed.

Unfortunately, pure synchronous systems exist only in theory. On the other hand, simply stating that every distributed system is asynchronous also does not do just to what we see in practice and we would be overly pessimistic in designing distributed systems under the assumption that they are necessarily asynchronous. Instead, it is more realistic to assume that a distributed system is **partially synchronous**: most of the time it behaves as a synchronous system, yet there is no bound on the time that it behaves in an asynchronous fashion. In other words, asynchronous behavior is an exception, meaning that we can normally use timeouts to conclude that a process has indeed crashed, but that occasionally such a conclusion is false.

In this context, halting failures can be classified as follows, from the least to the most severe (see also Cachin et al. [2011]). We let process P attempt to detect that process Q has failed.

- **Fail-stop failures** refer to crash failures that can be reliably detected. This may occur when assuming nonfaulty communication links and when the failure-detecting process P can place a worst-case delay on responses from Q.
- **Fail-noisy failures** are like fail-stop failures, except that P will only *eventually* come to the correct conclusion that Q has crashed. This means that there may be some a priori unknown time in which P's detections of the behavior of Q are unreliable.
- When dealing with **fail-silent failures**, we assume that communication links are nonfaulty, but that process P cannot distinguish crash failures from omission failures.
- **Fail-safe failures** cover the case of dealing with arbitrary failures by process Q, yet these failures are benign: they cannot do any harm.
- Finally, when dealing with **fail-arbitrary failures**, Q may fail in any possible way; failures may be unobservable in addition to being harmful to the otherwise correct behavior of other processes.

Clearly, having to deal with fail-arbitrary failures is the worst that can happen. As we shall discuss shortly, we can design distributed systems in such a way that they can even tolerate these types of failures.

8.1.3 Failure masking by redundancy

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy (see also Johnson [1995]). With **information redundancy**, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With **time redundancy**, an action is performed, and then, if need be, it is performed again. Transactions use this approach. If a transaction aborts, it can be redone with no harm. Another well-known example is retransmitting a request to a server when lacking an expected response. Time redundancy is especially helpful when the faults are transient or intermittent.

With **physical redundancy**, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. Physical redundancy can thus be done either in hardware or in software. For example, extra processes can be added to the system so that if a small number of them crash, the system can still function correctly. In other words, by replicating processes, a high degree of fault tolerance may be achieved. We return to this type of software redundancy below.

Note 8.3 (More information: Triple modular redundancy)
It is illustrative to see how redundancy has been applied in the design of electronic devices. Consider, for example, the circuit of Figure 8.3(a). Here signals pass through devices A, B, and C, in sequence. If one of them is faulty, the final result will probably be incorrect.

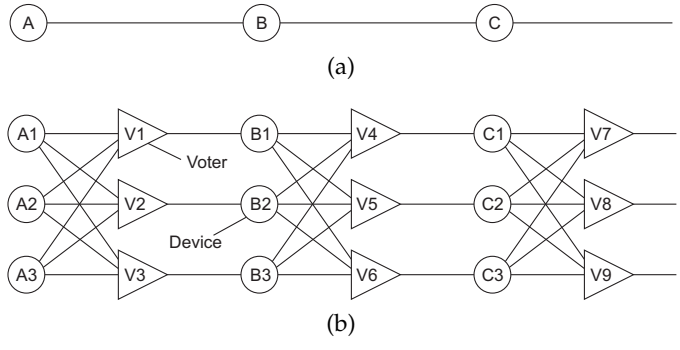

Figure 8.3: Triple modular redundancy.

In Figure 8.3(b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as **Triple Modular Redundancy** (**TMR**).

Suppose that element A_2 fails. Each of the voters, V_1, V_2, and V_3 gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage. In essence, the effect of A_2 failing is completely masked, so that the inputs to B_1, B_2, and B_3 are exactly the same as they would have been had no fault occurred.

Now consider what happens if B_3 and C_1 are also faulty, in addition to A_2. These effects are also masked, so the three final outputs are still correct.

At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass though the majority view. However, a voter is also a component and can also be faulty. Suppose, for example, that voter V_1 malfunctions. The input to B_1 will then be wrong, but as long as everything else works, B_2 and B_3 will produce the same output and V_4, V_5, and V_6 will all produce the correct result into stage three. A fault in V_1 is effectively no different than a fault in B_1. In both cases B_1 produces incorrect output, but in both cases it is voted down later and the final result is still correct.

Although not all fault-tolerant distributed systems use TMR, the technique is very general, and should give a clear feeling for what a fault-tolerant system is, as opposed to a system whose individual components are highly reliable but whose organization cannot tolerate faults (i.e., operate correctly even in the presence of faulty components). Of course, TMR can be applied recursively, for example, to make a chip highly reliable by using TMR inside it, unknown to the designers who use the chip, possibly in their own circuit containing multiple copies of the chips along with voters.

8.2 Process resilience

Now that the basic issues of fault tolerance have been discussed, let us concentrate on how fault tolerance can actually be achieved in distributed systems. The first topic we discuss is protection against process failures, which is achieved by replicating processes into groups. In the following pages, we consider the general design issues of process groups and discuss what a fault-tolerant group actually is. Also, we look at how to reach agreement within a process group when one or more of its members cannot be trusted to give correct answers.

8.2.1 Resilience by process groups

The key approach to tolerating a faulty process is to organize several identical processes into a group. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. In this way, if one process in a group fails, hopefully some other process can take over for it [Guerraoui and Schiper, 1997].

Process groups may be dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one during system operation. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

The purpose of introducing groups is to allow a process to deal with collections of other processes as a single abstraction. Thus a process P can send a message to a group $\mathbf{Q} = \{Q_1, \ldots, Q_N\}$ of servers without having to know who they are, how many there are, or where they are, which may change from one call to the next. To P, the group $\mathbf{Q}$ appears to be a single, logical process.

Group organization

An important distinction between different groups has to do with their internal structure. In some groups, all processes are equal. There is no distinctive leader and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there. More complex hierarchies are also possible, of course. These communication patterns are illustrated in Figure 8.4.

Each of these organizations has its own advantages and disadvantages. The flat group is symmetrical and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. For example, to decide anything, a vote often has to be taken, incurring some delay and overhead.

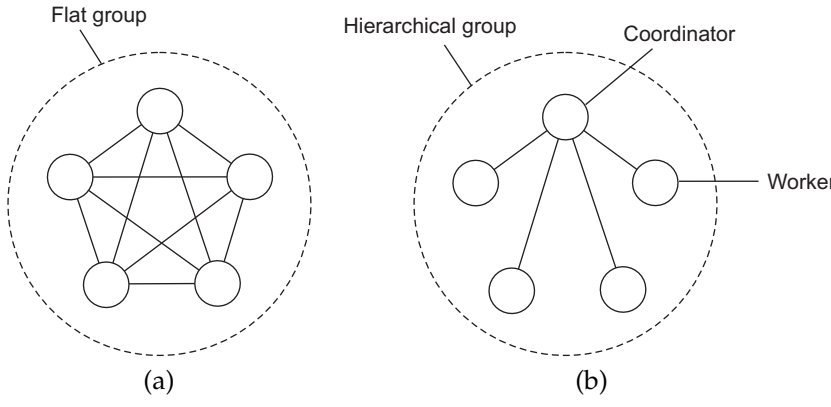

Figure 8.4: Communication in a (a) flat group and in a (b) hierarchical group.

The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else. In practice, when the coordinator in a hierarchical group fails, its role will need to be taken over and one of the workers is elected as new coordinator. We discussed leader-election algorithms in Chapter 6.

Membership management

When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups. One possible approach is to have a **group server** to which all these requests can be sent. The group server can then maintain a complete database of all the groups and their exact membership. This method is straightforward, efficient, and fairly easy to implement. Unfortunately, it shares a major disadvantage with all centralized techniques: a single point of failure. If the group server crashes, group management ceases to exist. Probably most or all groups will have to be reconstructed from scratch, possibly terminating whatever work was going on.

The opposite approach is to manage group membership in a distributed way. For example, if (reliable) multicasting is available, an outsider can send a message to all group members announcing its wish to join the group.

Ideally, to leave a group, a member just sends a goodbye message to everyone. In the context of fault tolerance, assuming fail-stop failure semantics is generally not appropriate. The trouble is, there is no polite announcement that a process crashes as there is when a process leaves voluntarily. The other members have to discover this experimentally by noticing that the crashed member no longer responds to anything. Once it is certain that the crashed member is really down (and not just slow), it can be removed from the group.

Another knotty issue is that leaving and joining have to be synchronous with

data messages being sent. In other words, starting at the instant that a process has joined a group, it must receive all messages sent to that group. Similarly, as soon as a process has left a group, it must not receive any more messages from the group, and the other members must not receive any more messages from it. One way of making sure that a join or leave is integrated into the message stream at the right place is to convert this operation into a sequence of messages sent to the whole group.

One final issue relating to group membership is what to do if so many processes go down that the group can no longer function at all. Some protocol is needed to rebuild the group. Invariably, some process will have to take the initiative to start the ball rolling, but what happens if two or three try at the same time? The protocol must to be able to withstand this. Again, coordination through, for example, a leader-election algorithm may be needed.

8.2.2 Failure masking and replication

Process groups are part of the solution for building fault-tolerant systems. In particular, having a group of identical processes allows us to mask one or more faulty processes in that group. In other words, we can replicate processes and organize them into a group to replace a single (vulnerable) process with a (fault tolerant) group. As discussed in the previous chapter, there are two ways to approach such replication: by means of primary-based protocols, or through replicated-write protocols.

Primary-based replication in the case of fault tolerance generally appears in the form of a primary-backup protocol. In this case, a group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations. In practice, the primary is fixed, although its role can be taken over by one of the backups, if need be. In effect, when the primary crashes, the backups execute some election algorithm to choose a new primary.

Replicated-write protocols are used in the form of active replication, as well as by means of quorum-based protocols. These solutions correspond to organizing a collection of identical processes into a flat group. The main advantage is that such groups have no single point of failure at the cost of distributed coordination.

An important issue with using process groups to tolerate faults is how much replication is needed. To simplify our discussion, let us consider only replicated-write systems. A system is said to be **k-fault tolerant** if it can survive faults in k components and still meet its specifications. If the components, say processes, fail silently, then having $k+1$ of them is enough to provide k-fault tolerance. If k of them simply stop, then the answer from the other one can be used.

On the other hand, if processes exhibit arbitrary failures, continuing to run when faulty and sending out erroneous or random replies, a minimum of $2 \cdot k + 1$ processes are needed to achieve k-fault tolerance. In the worst case, the k failing processes could accidentally (or even intentionally) generate the same reply. However, the remaining $k+1$ will also produce the same answer, so the client or voter can just believe the majority.

Now suppose that in a k-fault tolerant group a single process fails. The group as

a whole is still living up to its specifications, namely that it can tolerate the failure of up to k of its members (of which one has just failed). But what happens if more than k members fail? In that case all bets are off and whatever the group does, its results, if any, cannot be trusted. Another way of looking at this is that the process group, in its appearance of mimicking the behavior of a single, robust process, has failed.

8.2.3 Consensus in faulty systems

As mentioned, in terms of clients and servers, we have adopted a model in which a potentially very large collection of clients now send commands to a *group of processes* that jointly behave as a *single, highly robust process*. To make this work, we need to make an important assumption:

> *In a fault-tolerant process group, each nonfaulty process executes the same commands, and in the same order, as every other nonfaulty process.*

Formally, this means that the group members need to reach **consensus** on which command to execute. If failures cannot happen, reaching consensus is easy. For example, we can use Lamport's totally ordered multicasting as described in Section 6.2.1. Or, to keep it simple, using a centralized sequencer that hands out a sequence number to each command that needs to be executed will do the job as well. Unfortunately, life is not without failures, and reaching consensus among a group of processes under more realistic assumptions turns out to be tricky.

To illustrate the problem at hand, let us assume we have a group of processes $\mathbf{P} = \{P_1, \ldots, P_n\}$ operating under fail-stop failure semantics. In other words, we assume that crash failures can be reliably detected among the group members. Typically a client contacts a group member requesting it to execute a command. Every group member maintains a list of proposed commands: some which it received directly from clients; others which it received from its fellow group members. We can reach consensus using the following approach, adopted from Cachin et al. [2011], and referred to as **flooding consensus**.

The algorithm operates in rounds. In each round, a process P_i sends its list of proposed commands it has seen so far to every other process in $\mathbf{P}$. At the end of a round, each process merges all received proposed commands into a new list, from which it then will deterministically select the command to execute, if possible. It is important to realize that the selection algorithm is the same for all processes. In other words, if all process have exactly the same list, they will all select the same command to execute (and remove that command from their list).

It is not difficult to see that this approach works as long as processes do not fail. Problems start when a process P_i detects, during round r, that, say process P_k has crashed. To make this concrete, assume we have a process group of four processes $\{P_1, \ldots, P_4\}$ and that P_1 crashes during round r. Also, assume that P_2 receives the list of proposed commands from P_1 before it crashes, but that P_3 and P_4 do not (in other words, P_1 crashes before it got a chance to send its list to P_3 and P_4). This situation is sketched in Figure 8.5.

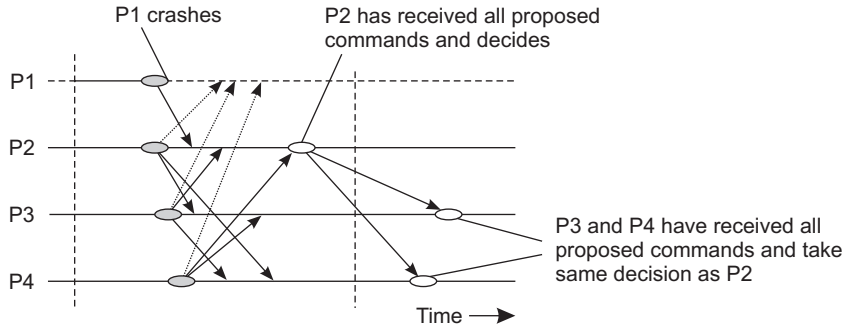

Figure 8.5: Reaching consensus through flooding in the presence of crash failures. Adopted from Cachin et al. [2011].

Assuming that all processes knew who was group member at the beginning of round r, P_2 is ready to make a decision on which command to execute when it receives the respective lists of the other members: it has all commands proposed so far. Not so for P_3 and P_4. For example, P_3 may detect that P_1 crashed, but it does not know if either P_2 or P_4 had already received P_1's list. From P_3's perspective, if there is another process that did receive P_1's proposed commands, that process may then make a different decision then itself. As a consequence, the best that P_3 can do is postpone its decision until the next round. The same holds for P_4 in this example. A process will decide to move to a next round when it has received a message from every nonfaulty process. This assumes that each process can reliably detect the crashing of another process, for otherwise it would not be able to decide who the nonfaulty processes are.

Because process P_2 received all commands, it can indeed make a decision and can subsequently broadcast that decision to the others. Then, during the next round $r + 1$, processes P_3 and P_4 will also be able to make a decision: they will decide to execute the same command selected by P_2.

To understand why this algorithm is correct, it is important to realize that a process will move to a next round without having made a decision, only when it detects that another process has failed. In the end, this means that in the worst case at most one nonfaulty process remains, and this process can simply decide whatever proposed command to execute. Again, note that we are assuming reliable failure detection.

But then, what happens when the decision by process P_2 that it sent to P_3 was lost? In that case, P_3 can still not make a decision. Worse, we need to make sure that it makes the same decision as P_2 and P_4. If P_2 did not crash, we can assume that a retransmission of its decision will save the day. If P_2 did crash, this will be also detected by P_4 who will then subsequently rebroadcast its decision. In the meantime, P_3 has moved to a next round, and after receiving the decision by P_4, will terminate its execution of the algorithm.

Essential Paxos

The flooding-based consensus algorithm is not very realistic if only for the fact that it relies on a fail-stop failure model. More realistic is to assume a fail-noisy failure model in which a process will *eventually* reliably detect that another process has crashed. In the following, we describe a simplified version of a widely adopted consensus algorithm, known as **Paxos**. It was originally published in 1989 as a technical report by Leslie Lamport, but it took about a decade before someone decided that it may not be such a bad idea to disseminate it through a regular scientific channel [Lamport, 1998]. The original publication is not easy to understand, exemplified by other publications that aim at explaining it [Lampson, 1996; Prisco et al., 1997; Lamport, 2001]. In the following, we will stick to the essence of Lamport's original proposal.

The assumptions under which Paxos operates are rather weak:

- The distributed systems is partially synchronous (in fact, it may even be asynchronous).
- Communication between processes may be unreliable, meaning that messages may be lost, duplicated, or reordered.
- Messages that are corrupted can be detected as such (and thus subsequently ignored).
- All operations are deterministic: once an execution is started, it is known exactly what it will do.
- Processes may exhibit crash failures, but not arbitrary failures, nor do processes collude.

By-and-large, these are realistic assumptions for many practical distributed systems.

We follow the explanation given by Lamport [2001] to build an understanding of the Paxos algorithm. The algorithm operates as a network of communicating processes, of which there are different types. First, there are **clients** that request a specific operation to be executed. At the server side, each client is represented by a single **proposer**, which is a process that will attempt to have a client's request accepted. There may be several proposers, each representing one or more clients. What we need to establish is that a proposed operation is accepted by an **acceptor**. If a majority of acceptors accepts the same proposal, the proposal is said to be *chosen*. However, what is chosen still needs to be *learned*. To this end, we will have a number of **learner** processes, each of which will execute a chosen proposal once it has been informed by a majority of acceptors.

A single proposer, acceptor, and learner are grouped together to form the logical server, running on a single machine, that the client communicates with, as shown in Figure 8.6. By replicating this server we aim at obtaining fault tolerance in the presence of crash failures.

The basic model is that a proposer makes a proposal (on behalf of a client) by sending its proposal to all acceptors. As there may be multiple concurrent proposals sent roughly at the same time, we first need to make sure that different proposals

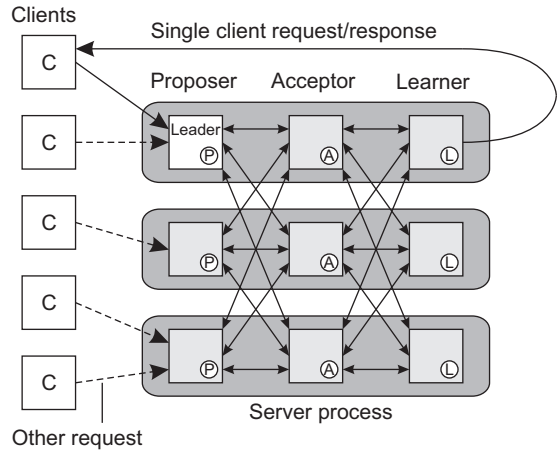

Figure 8.6: The organization of Paxos into different processes.

can be distinguished from one another. Therefore, each proposal p has a uniquely associated number $id(p)$. How uniqueness is achieved is left to an implementation. Second, to make sure that a group of acceptors can actually choose a proposal, we need to allow an acceptor to accept multiple proposals (and thus that it can change its initial choice).

Let $oper(p)$ denote the operation associated with proposal p. The trick is to allow multiple proposals to be accepted, but that each of these proposals has the same associated operation. This can be achieved by guaranteeing that if a proposal p is chosen, that any higher-numbered proposal will also have the same associated operation. In other words, we require that

$$p \text{ is chosen } \Rightarrow \text{ for all } p' \text{ with } id(p') > id(p) : oper(p') = oper(p)$$

Of course, for p to be chosen, it needs to be accepted. That means that we can guarantee our requirement when guaranteeing that if p is chosen, then any higher-numbered proposal accepted by any acceptor, has the same associated operation as p. However, this is not sufficient, for suppose that at a certain moment a proposer simply sends a new proposal p', with the highest number so far, to an acceptor A that had not received any proposal before (which may happen according to our assumptions concerning message loss). In absence of any other proposals, A will simply accept p'. To prevent this situation from happening, we thus need to guarantee that

> *If proposal p is chosen, then any higher-numbered proposal issued by a proposer, has the same associated operation as p.*

When explaining the Paxos algorithm below, we will indeed see that a proposer may need to adopt an operation coming from acceptors in favor of its own.

The processes collectively formally ensure *safety*, in the sense that only proposed operations will be learned, and that at most one operation will be learned at a time. Furthermore, Paxos ensures *conditional liveness* in the sense that if enough processes remain up-and-running, then a proposed operation will eventually be learned (and thus executed). Liveness in general is not guaranteed, unless some small adaptations are made.

There are two phases, each in turn consisting of two subphases. During the first phase, a proposer interacts with acceptors to get a requested operation accepted for execution. The best that can happen is that an individual acceptor promises to consider the proposer's operation and ignore other requests. The worst is that the proposer was too late and that it will be asked to adopt some other proposer's request instead.

In the second phase, the acceptors will have informed proposers about the promises they have made. Proposers essentially take up a slightly different role by promoting a single operation to the one to be executed, and subsequently telling the acceptors.

Phase 1a (prepare): The goal of this phase is that a proposer PP who is proposing operation o, tries to get its proposal number **anchored**, in the sense that any lower number failed, or that o had also been previously proposed (i.e., with some lower proposal number). To this end, PP communicates with a quorum of acceptors. For the operation o, the proposer selects a counter m higher than any of its previously selected counters. This leads to a **proposal number** $r = (m, i)$ where i is the (numerical) process identifier of PP. Note that

$$(m, i) < (n, j) \Leftrightarrow (m < n) \text{ or } (m = n \text{ and } i < j)$$

Proposer PP sends $prepare(r)$ to a majority of acceptors. In doing so, it is (1) asking the acceptors to promise not to accept any proposals with a lower proposal number, and (2) to inform it about an accepted proposal, if any, with the highest number less than r. Note that if such a proposal p exists, the proposer will adopt the associated operation $oper(p)$.

Phase 1b (promise): An acceptor PA receives multiple proposals. Assume it receives $prepare(r)$ from PP. There are three cases to consider:

- r is the highest proposal number received from any proposer so far. In that case, PA will return a promise $promise(r)$ to PP stating that PA will ignore any future proposals with a lower proposal number.
- If r is the highest number so far, but another proposal (r', o') had already been accepted, PA also returns (r', o') to PP. This will allow PP to decide on the final operation that needs to be accepted.
- In all other cases, do nothing: there is apparently another proposal with a higher proposal number that is being processed.

Once the first phase has been completed, proposers know what the acceptors have promised. This will put a proposer into a position to tell the acceptors what to accept:

Phase 2a (accept): There are two cases to consider:

- If a proposer PP does not receive any accepted operation from any of the acceptors, it will forward its own proposal for acceptance by sending $accept(r, o)$ to a majority of acceptors.
- Otherwise, it was informed about another operation o', which it will adopt and forward for acceptance by sending $accept(r, o')$, where r is the proposer's proposal number and o' is the operation with proposal number highest among all accepted operations that were returned by the acceptors in Phase 1b.

Phase 2b (learn): Finally, if an acceptor PA receives $accept(r, o')$, but did not previously send a promise with a higher proposal number, it will accept operation o' and tell all learners by sending $learn(o')$. A learner PL receiving $learn(o')$ from a majority of acceptors, will execute the operation o'.

8.2.4 Failure detection

It may have become clear from our discussions so far that in order to properly mask failures, we generally need to detect them as well. Failure detection is one of the cornerstones of fault tolerance in distributed systems. What it all boils down to is that for a group of processes, nonfaulty members should be able to decide who is still a member, and who is not. In other words, we need to be able to detect when a member has failed.

When it comes to detecting process failures, there are essentially only two mechanisms. Either processes actively send "are you alive?" messages to each other (for which they obviously expect an answer), or passively wait until messages come in from different processes. The latter approach makes sense only when it can be guaranteed that there is enough communication between processes.

There has been a huge body of theoretical work on failure detectors. What it all boils down to is that a timeout mechanism is used to check whether a process has failed. If a process P probes another process Q to see if has failed, P is said to **suspect** Q to have crashed if Q has not responded within some time.

> **Note 8.4** (More information: On perfect failure detectors)
> It should be clear that in a synchronous distributed system, a suspected crash corresponds to a known crash. In practice, however, we will be dealing with partially synchronous systems. In that case, it makes more sense to assume **eventually perfect failure detectors**. In this case, a process P will suspect another process Q to have crashed after t time units have elapsed and still Q did not respond to P's probe. However, if Q later does send a message that is (also) received by P, P will (1) stop suspecting Q, and (2) increase

the timeout value t. Note that if Q does crash (and does not recover), P will continue to suspect Q.

In real settings, there are problems with using probes and timeouts. For example, due to unreliable networks, simply stating that a process has failed because it does not return an answer to a probe message may be wrong. In other words, it is quite easy to generate false positives. If a false positive has the effect that a perfectly healthy process is removed from a membership list, then clearly we are doing something wrong. Another serious problem is that timeouts are just plain crude. As noticed by Birman [2012], there is hardly any work on building proper failure detection subsystems that take more into account than only the lack of a reply to a single message. This statement is even more evident when looking at industry-deployed distributed systems.

There are various issues that need to be taken into account when designing a failure detection subsystem [see also Zhuang et al. [2005]]. For example, failure detection can take place through gossiping in which each node regularly announces to its neighbors that it is still up and running. As we mentioned, an alternative is to let nodes actively probe each other.

Failure detection can also be done as a side-effect of regularly exchanging information with neighbors, as is the case with gossip-based information dissemination (which we discussed in Chapter 4). This approach is essentially also adopted in Obduro [Vogels, 2003]: processes periodically gossip their service availability. This information is gradually disseminated through the network by gossiping. Eventually, every process will know about every other process, but more importantly, will have enough information locally available to decide whether a process has failed or not. A member for which the availability information is old, will presumably have failed.

Another important issue is that a failure detection subsystem should ideally be able to distinguish network failures from node failures. One way of dealing with this problem is not to let a single node decide whether one of its neighbors has crashed. Instead, when noticing a timeout on a probe message, a node requests other neighbors to see whether they can reach the presumed failing node. Of course, positive information can also be shared: if a node is still alive, that information can be forwarded to other interested parties (who may be detecting a link failure to the suspected node).

This brings us to another key issue: when a member failure is detected, how should other nonfaulty processes be informed? One simple, and somewhat radical approach is the one followed in FUSE [Dunagan et al., 2004]. In FUSE, processes can be joined in a group that spans a wide-area network. The group members create a spanning tree that is used for monitoring member failures. Members send ping messages to their neighbors. When a neighbor does not respond, the pinging node immediately switches to a state in which it will also no longer respond to pings from

other nodes. By recursion, it is seen that a single node failure is rapidly promoted to a group failure notification. FUSE does not suffer a lot from link failures for the simple reason that it relies on point-to-point TCP connections between group members.