7.1 INTRODUCTION TO FAULT TOLERANCE

Fault tolerance is 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).

7.1.1 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):

1. Availability
2. Reliability
3. Safety
4. 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 for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable. Similarly, a system that never crashes but is shut down for two 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, nothing catastrophic 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 easy 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.

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 not an option.

Building dependable systems closely relates to controlling faults. A distinction is made between preventing, removing, and forecasting faults (Laprie, 1995). For our purposes, the most important issue is **fault tolerance**, meaning that a system can provide its services even in the presence of faults.

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, whenever the fault doctor shows up, the system works fine.

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

7.1.2 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 each other 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 actually 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 Fig. 7-1, 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 *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	A server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure	A server may produce arbitrary responses at arbitrary times

Figure 7-1. Different types of failures.

A **crash failure** occurs when a server prematurely halts, but was working correctly until it stopped. An important aspect with 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. Many personal computer systems suffer from crash failures so often that people have come to expect them to be normal. In this sense, 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, the

server perhaps 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 may need to be prepared that the client will 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 Chap. 2, 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 provides the wrong reply to a request. For example, a search engine that systematically returns Web pages not related to any of the used search terms, 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. Worse yet a faulty server may even be maliciously working together with other servers to produce intentionally wrong answers. This situation illustrates why security is also considered an important requirement when talking about dependable systems. 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.) Byzantine faults were first analyzed by Pease et al. (1980) and Lamport et al. (1982). We return

to such failures below.

Arbitrary failures are closely related to crash failures. The definition of crash failures as presented above is the most benign way for a server to halt. They are also referred to as **fail-stop failures**. In effect, a fail-stop server will simply stop producing output in such a way that its halting can be detected by other processes. For example, the server may have been so friendly to announce it is about to crash.

Of course, in real life, servers halt by exhibiting omission or crash failures, and are not so friendly as to announce they are going to stop. It is up to the other processes to decide that a server has prematurely halted. However, in such **fail-silent systems**, the other process may incorrectly conclude that a server has halted. Instead, the server may just be unexpectedly slow, that is, it is exhibiting performance failures.

Finally, there are also occasions in which the server is producing random output, but this output can be recognized by other processes as plain junk. The server is then exhibiting arbitrary failures, but in a benign way. These faults are also referred to as being **fail-safe**.

7.1.3 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. Using transactions (described in Chap. 5) is an example of this approach. If a transaction aborts, it can be redone with no harm. 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.

Physical redundancy is a well-known technique for providing fault tolerance. It is used in biology (mammals have two eyes, two ears, two lungs, etc.), aircraft (747s have four engines but can fly on three), and sports (multiple referees in case one misses an event). It has also been used for fault tolerance in electronic circuits

for years; it is illustrative to see how it has been applied there. Consider, for example, the circuit of Fig. 7-2(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.

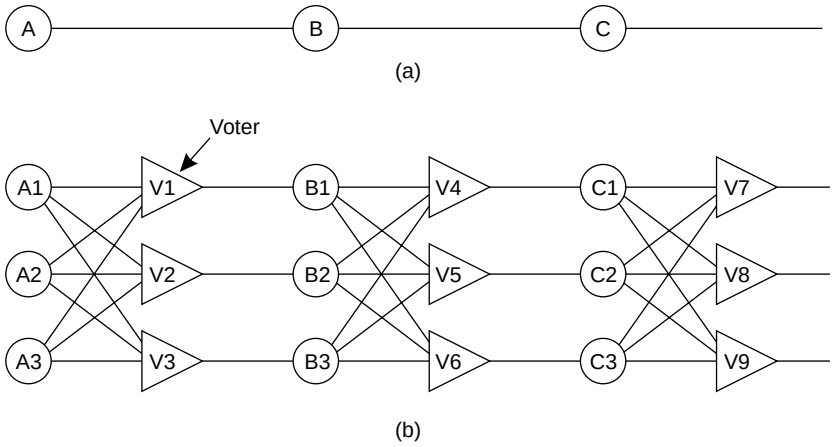


Figure 7-2. Triple modular redundancy.

In Fig. 7-2(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 **TMR** (**Triple Modular Redundancy**).

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 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.

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 is not fault tolerant. 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.