than a 1. A similar distinction exists between programming mistakes and execution errors. Consider, for example, a subroutine that is supposed to compute sin(x), but owing to a programming mistake calculates the absolute value of sin(x) instead. This mistake will result in an execution error only if that particular subroutine is used and the correct result is negative.

Both faults and errors can spread through the system. For example, if a chip shorts out power to ground, it may cause nearby chips to fail as well. Errors can spread when the output of one unit is used as input by other units. To return to our previous examples, the erroneous results of either the faulty adder or the sin(x) subroutine can be fed into further calculations, thus propagating the error.

To limit such contagion, designers incorporate *containment zones* into systems. These are barriers that reduce the chance that a fault or error in one zone will propagate to another. For example, a fault-containment zone can be created by ensuring that the maximum possible voltage swings in one zone are insulated from the other zones, and by providing an independent power supply to each zone. In other words, the designer tries to electrically isolate one zone from another. An error-containment zone can be created, as we will see in some detail later on, by using redundant units/programs and voting on their output.

Faults can be classified along multiple dimensions. Some important dimensions are: their duration, when they were introduced, whether or not there was conscious intent behind their introduction, and whether they occurred in hardware or in software. Let us look at each of these dimensions in turn.

Duration: Duration is an important classification dimension for hardware faults. These can be classified into *permanent*, *transient*, or *intermittent*. A *permanent fault* is just that: it reflects a component going out of commission permanently. As an example of a permanent fault, think of a burned-out lightbulb. A *transient fault* is one that causes a component to malfunction for some time; it goes away after that time, and the functionality of the component is fully restored. As an example, think of a random noise interference during a telephone conversation. Another example is a memory cell with contents that are changed spuriously due to some electromagnetic interference. The cell itself is undamaged: it is just that its contents are wrong for the time being, and overwriting the memory cell will make the fault go away. An *intermittent fault* never quite goes away entirely; it oscillates between being quiescent and active. When the fault is quiescent, the component functions normally; when the fault is active, the component malfunctions. An example for an intermittent fault is a loose electrical connection.

When they were introduced: Faults can be introduced in various phases of the system's lifetime. Faulty design decisions cause faults to be introduced in the design phase. They can be born during system implementation (e.g., software development). They can occur during system operation due to hardware degradation, faulty software updates, or harsh environments (e.g., due to high levels of radiation or excessive temperatures).

Intent: Faults may be intentional or unintentional. Most software bugs are unintentional faults. For example, consider the Fortran instruction `doi=1.35` when the programmer meant to type `do i=1,35`. The programmer was trying to set up a loop; what the system saw was an instruction to assign the value of 1.35 to the variable `doi`.

They may be intentional: consciously undertaken design decisions may lead to faults in the system design. Such faults can be further subclassified into *nonmalicious* and *malicious* categories. Nonmalicious design faults are introduced with the best of intentions; often because of side-effects (often in the way that various modules of the system interact) that were not foreseen at design time, or because the operating environment was imperfectly understood. Malicious faults are introduced with malicious intent: for instance, a programmer may deliberately insert a weak point in software that permits unauthorized access to some data structure. We also include in this subcategory faults, which may not be deliberately introduced, but which behave as if they were the product of nefarious intent. For example, a component may fail in such a way that it "acts as if malicious" and sends differently valued outputs to different receivers. Think of an altitude sensor in an airplane that reports a 1000-foot altitude to one unit, while reporting an 8000-foot altitude to another unit. Malicious faults are also known as Byzantine faults.

1.2 TYPES OF REDUNDANCY

All of fault tolerance is an exercise in exploiting and managing *redundancy*. Redundancy is the property of having more of a resource than is minimally necessary to do the job at hand. As failures happen, redundancy is exploited to mask or otherwise work around these failures, thus maintaining the desired level of functionality.

There are four forms of redundancy that we will study: hardware, software, information, and time. Hardware faults are usually dealt with by using hardware, information, or time redundancy, whereas software faults (bugs) are mostly protected against by software redundancy.

Hardware redundancy is provided by incorporating extra hardware into the design to either detect or override the effects of a failed component. For example, instead of having a single processor, we can use two or three processors, each performing the same function. By having two processors and comparing their results, we can detect the failure of a single processor; by having three, we can use the majority output to override the wrong output of a single faulty processor. This is an example of *static hardware redundancy*, the main objective of which is the immediate masking of a failure. A different form of hardware redundancy is *dynamic redundancy,* where spare components are activated upon the failure of a currently active component. A combination of static and dynamic redundancy techniques is also possible, leading to *hybrid hardware redundancy*.

Hardware redundancy can thus range from a simple duplication to complicated structures that switch in spare units when active ones become faulty. These forms of hardware redundancy incur high overheads, and their use is therefore normally reserved for critical systems, where such overheads can be justified. In particular, substantial amounts of redundancy are required to protect against malicious faults.

The best-known form of information redundancy is error detection and correction coding. Here, extra bits (called check bits) are added to the original data bits so that an error in the data bits can be detected or even corrected. The resulting error-detecting and error-correcting codes are widely used today in memory units and various storage devices to protect against benign failures. Note that these error codes (like any other form of information redundancy) may require extra hardware to process the redundant data (the check bits).

Error-detecting and error-correcting codes are also used to protect data communicated over noisy channels, which are channels that are subject to many transient failures. These channels can be the communication links among either widely separated processors (e.g., the Internet) or processors that form a local network. If the code used for data communication is capable of only detecting the faults that have occurred (but not correcting them), we can retransmit as necessary, thus employing time redundancy.

In addition to transient data communication failures due to noise, local and wide-area networks may experience permanent link failures. These failures may disconnect one or more existing communication paths, resulting in a longer communication delay between certain nodes in the network, a lower data bandwidth between certain node pairs, or even a complete disconnection of certain nodes from the rest of the network. Redundant communication links (i.e., hardware redundancy) can alleviate these problems.

Computing nodes can also exploit time redundancy through reexecution of the same program on the same hardware. As before, time redundancy is effective against *transient* faults. Because the majority of hardware faults are transient, it is unlikely that the separate executions (if spaced sufficiently apart) will experience the same fault. Time redundancy can thus be used to detect transient faults in situations, where such faults may otherwise go undetected. Time redundancy can also be used when other means for detecting errors are in place, and the system is capable of recovering from the effects of the fault and repeating the computation. Compared with the other forms of redundancy, time redundancy has much lower hardware and software overhead, but incurs a high-performance penalty.

Software redundancy is used mainly against software failures. It is a reasonable guess that *every* large piece of software that has ever been produced has contained faults (bugs). Dealing with such faults can be expensive: one way is to independently produce two or more versions of that software (preferably by disjoint teams of programmers) in the hope that the different versions will not fail on the same input. The secondary version(s) can be based on simpler and less accurate algorithms (and, consequently, less likely to have faults) to be used only upon the failure of the primary software to produce acceptable results. Just as for hardware redundancy, the multiple versions of the program can be executed either concurrently (requiring redundant hardware as well) or sequentially (requiring extra time, i.e., time redundancy) upon a failure detection.

1.3 BASIC MEASURES OF FAULT TOLERANCE

Since fault tolerance is about making machines more dependable, it is important to have proper measures (yardsticks) by which to gauge such dependability. In this section, we will examine some of these yardsticks and their application.

A measure is a mathematical abstraction, that expresses some relevant facet of the performance of its object. By its very nature, a measure only captures some subset of the properties of its object. The trick in defining a suitable measure is to keep this subset large enough so that behaviors of interest to the user are captured, and yet not so large that the measure loses focus.

1.3.1 TRADITIONAL MEASURES

We first describe the traditional measures of dependability of a single computer. These metrics have been around for a long time, and measure very basic attributes of the system. Two of these measures are *reliability* and *availability*.

The conventional definition of reliability, denoted by $R(t)$, is the probability (as a function of the time t) that the system has been up continuously in the time interval $[0, t]$, conditioned on the event that it was up at time 0. This measure is suitable for applications, in which even a momentary disruption can prove costly. One example is computers that control physical processes such as an aircraft, for which system-wide computer failure could result in catastrophe.

Closely related to reliability are the *mean time to failure,* denoted by MTTF, and the *mean time between failures,* MTBF. The first is the average time the system operates until a failure occurs, whereas the second is the average time between two consecutive failures. The difference between the two is due to the time needed to repair the system following the first failure. Denoting the *mean time to repair* by MTTR, we obtain

$$\mathsf{MTBF} = \mathsf{MTTF} + \mathsf{MTTR}.$$

Availability, denoted by $A(t)$, is the average fraction of time over the interval $[0, t]$ that the system is up. This measure is appropriate for applications, in which continuous fault-free operation is not vital, but where it would be expensive to have the system down for a significant amount of time. An airline reservation system needs to be highly available, because downtime can put off customers and lose sales; however, an occasional (very) short-duration failure can be tolerated.

The long-term availability, denoted by A, is defined as

$$A = \lim_{t \to \infty} A(t).$$

A can be interpreted as the probability that the system will be up at some random point in time, and is meaningful only in systems that include repair of faulty components. The long-term availability can be calculated from MTTF, MTBF, and MTTR as follows:

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

A related measure, *point availability*, denoted by $A_p(t)$, is the probability that the system is up at the particular time instant t.

It is possible for a low-reliability system to have high availability: consider a system that fails every hour on the average, but comes back up after only a second. Such a system has an MTBF of just 1 hour and, consequently, a low reliability; however, its availability is high: $A = 3599/3600 = 0.99972$.

These definitions assume, of course, that we have a state, in which the system can be said to be “up”, and another in which it is not. For simple components, this is a good assumption. For example, a lightbulb is either good or burned out. A wire is either connected or has a break in it. However, for even simple systems, such an assumption can be very limiting. For example, consider a processor that has one of its several hundreds of millions of gates stuck at logic value 0. In other words, the output of this logic gate is always 0, regardless of the input. Suppose the rest of the processor is functional, and that this failed logic gate only affects the output of the processor about once in every 25,000 hours of use. For example, a particular gate in the divide unit when faulty may result in a wrong quotient if the divisor is within a certain subset of values. Clearly, the processor is not fault-free, but would one define it as “down”?

The same remarks apply with even greater force to systems that degrade gracefully. By this, we mean systems with various levels of functionality. Initially, with all of its components operational, the system is at its highest level of functionality. As these components fail, the system degrades from one level of functionality to the next. Beyond a certain point, the system is unable to produce anything of use and fails completely. As with the previous example, the system has multiple “up” states. Is it said to fail when it degrades from full to partial functionality? Or when it fails to produce any useful output at all? Or when its functionality falls below a certain threshold? If the last, what is this threshold, and how is it determined?

We can therefore see that traditional reliability and availability are very limited in what they can express. There are obvious extensions to these measures. For example, we may consider the average computational capacity of a system with n processors. Let c_i denote the computational capacity of a system with i operational processors. This can be a simple linear function of the number of processors, $c_i = ic_1$, or a more complex function of i, depending on the ability of the application to utilize i processors. The average computational capacity of the system at time t can then be defined as $\sum_{i=1}^{n} c_i P_i(t)$, where $P_i(t)$ is the probability that exactly i processors are operational at time t. In contrast, the point availability of the system at time t will be

$$A_p(t) = \sum_{i=m}^{n} P_i(t),$$

where m is the minimum number of processors necessary for proper operation of the system.

1.3.2 NETWORK MEASURES

In addition to the general system measures previously discussed, there are also more specialized measures, focusing on the network that connects the processors together. The simplest of these are classical node and line *connectivity*, which are defined as the minimum number of nodes and lines, respectively, that have to fail before the network becomes disconnected. This gives a rough indication of how vulnerable a network is to disconnection. For example, a network that can be disconnected by the failure of just one (critically positioned) node is potentially more vulnerable than another that requires at least four nodes to fail before it becomes disconnected.

Classical connectivity is a very basic measure of network reliability. Like reliability, it distinguishes between only two network states: connected and disconnected. It says nothing about how the network degrades as nodes fail before, or after, becoming disconnected. Consider the two networks shown in Fig. 1.1. Both networks have the same classical node connectivity of 1. However, in a real sense, network N1 is much more "connected" than N2. The probability that N2 splinters into small pieces is greater than that for N1.

To express this type of "connectivity robustness," we can use additional measures. Two such measures are the average node-pair distance, and the network diameter (the maximum node-pair distance), both calculated given the probability of node and/or link failure. Such network measures, together with the traditional measures listed above, allow us to gauge the dependability of various networked systems that consist of computing nodes connected through a network of communication links.

1.4 OUTLINE OF THIS BOOK

The next chapter is devoted to hardware fault tolerance. This is the most established topic within fault-tolerant computing, and many of the basic principles and techniques that have been developed

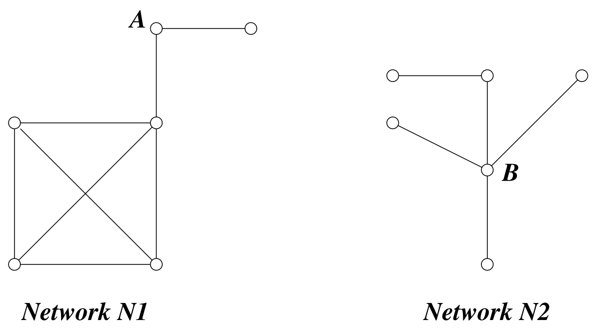

FIGURE 1.1

Inadequacy of classical connectivity.

for it have been extended to other forms of fault tolerance. Prominent hardware failure mechanisms are discussed, as well as canonical fault-tolerant redundant structures. The notion of Byzantine, or malicious, failure is introduced. Techniques to evaluate the reliability and availability of fault-tolerant systems are introduced here, including the use of Markov models.

Next, several variations of information redundancy are covered, starting with the most widely used error-detecting and error-correcting codes. Then, other forms of information redundancy are discussed, including storage redundancy (RAID systems), data replication in distributed systems, and the algorithm-based fault-tolerance technique that tolerates data errors in array computations using some error-detecting and error-correcting codes.

Many computing systems nowadays consist of multiple networked processors that are subject to interconnection link failures, in addition to the already-discussed single node/processor failures. We therefore present in this book suitable fault tolerance techniques for these networks and analysis methods to determine which network topologies are more robust. With the number of transistors on a chip increasing in every hardware generation, networks-on-chip are becoming commonplace. Another recent development is the proliferation of sensor networks. We discuss fault-tolerance techniques for both of these.

Software mistakes/bugs are, in practice, unavoidable, and consequently, some level of software fault tolerance is often necessary. This can be as simple as acceptance tests to check the reasonableness of the results before using them, or as complex as running two or more versions of the software (sequentially or in parallel). Programs also tend to have their state deteriorate after running for long periods of time and eventually crash. This situation can be avoided by periodically restarting the program, a process called rejuvenation. Hypervisor-based systems, where a hardware platform supports multiple virtual machines, each running its own operating system, have gained popularity; we discuss fault-tolerance issues associated with these. Finally, there is the issue of modeling software reliability. Unlike hardware faults, software bugs are very hard to model. Still, a few such models have been developed and several of them are described.

Hardware fault-tolerance techniques can be quite costly to implement. In applications, in which a complete and immediate masking of the effect of hardware faults (especially of a transient nature) is not necessary, checkpointing is an inexpensive alternative. For programs that run for a long time and for which reexecution upon a failure might be too costly, the program state can be saved (once or periodically) during the execution. Upon a failure occurrence, the system can roll back the program to the most recent checkpoint and resume its execution from that point. Checkpointing is especially important in exascale computing, where the computing base may consist of thousands of processors jointly executing programs that take hours, days, or even weeks to execute. Various checkpointing techniques are presented and analyzed in this book, for both general-purpose computing and real-time systems.

Cyber-physical systems, which consist of physical *plants* controlled by computer, have taken off in recent years. Examples include fly-by-wire aircraft, automobiles, spacecraft, power grids, chemical reactors, and intelligent highway systems. Such applications are often life-critical and must therefore be highly reliable. They consist of sensors to assess the state of the plant and the operating environment, computers to run the control software, and actuators to impose their control outputs on the plant. Fault-tolerance issues associated with each are discussed in this book.

Next is a chapter consisting of a few case studies, which serve to illustrate the use of many of the fault-tolerance techniques described previously.

An important part of the design and evaluation process of a fault-tolerant system is to demonstrate that the system does indeed function at the advertised level of reliability. Often, the designed system is too complex to develop analytical expressions of its reliability. If a prototype of the system has already been constructed, then fault-injection experiments can be performed and certain dependability attributes measured. If, however, as is very common, a prototype does not yet exist, statistical simulation may be the only option. Simulation programs for complex systems must be carefully designed to produce accurate results without requiring excessive amounts of computation time. We discuss the principles that should be followed when preparing a simulation program, and show how simulation results can be analyzed to infer system reliability.

We end the book with two specialized fault-tolerance topics: defect tolerant VLSI (very large-scale integration) design and fault tolerance in cryptographic devices. The increasing complexity of VLSI chip design has resulted in a situation, in which manufacturing defects are unavoidable. If nothing is done to remedy this situation, the expected *yield* (the fraction of manufactured chips, which are operational) will be very low. Thus techniques to reduce the sensitivity of VLSI chips to defects have been developed, some of which are very similar to the hardware redundancy schemes.

For cryptographic devices, the need for fault tolerance is twofold. Not only is it crucial that such devices (e.g., smart cards) operate in a fault-free manner in whatever environment they are used, but more importantly, they must stay secure. Fault-injection-based attacks on cryptographic devices have become the simplest and fastest way to extract the secret key from the device. Thus the incorporation of fault tolerance can help to keep cryptographic devices secure.

1.5 FURTHER READING

Several textbooks and reference books on the topic of fault tolerance are available. See, for example, [5–7,10,14,18–20]. Fault tolerance from a software perspective is treated in [3,12,15]. Excellent classifications of the various approaches taken can be found in [2,17]. The major conference in the field is the *Conference on Dependable Systems and Networks* (DSN) [4]; this is a successor to the *Fault-Tolerant Computing Symposium* (FTCS).

Fault tolerance in specific applications are treated in a number of works; for instance embedded/cyber-physical systems in [1,8], cloud computing [9], and high-performance computing [13].

The concept of computing being invisible everywhere appeared in [22], in the context of *pervasive computing*, i.e., computing which pervades everyday living, without being obtrusive.

The definitions of the basic terms and measures appear in most of the textbooks mentioned above and in several probability and statistics books. For example, see [21]. Our definitions of fault and error are slightly different from those used in some of the references. One definition of an error is that it is that part of the system state that leads to system failure. Strictly interpreted, this only applies to a system with state, i.e., with memory. We use the more encompassing definition of anything that can be construed as a manifestation of a fault. This wider interpretation allows purely combinational circuits, which are stateless, to generate errors.

One measure of dependability that we did not describe in the text is to consider everything from the perspective of the application. This approach was taken to define the measure known as *performability*. The application is used to define "accomplishment levels" $L_1, L_2, \cdots, L_n$. Each of these represents a level of quality of service delivered by the application. Now, the performance of the computer affects this quality (if it did not, by definition, it would have nothing to do with the application!). The approach taken by performability is to link the performance of the computer to the accomplishment level that this enables. Performability is then a vector, $(P(L_1), P(L_2), \cdots, P(L_n))$, where $P(L_i)$ is the probability that the computer functions well enough to permit the application to reach up to accomplishment level L_i. For more on performability, see [11,16].

REFERENCES

[1] I. Alvarez, A. Ballesteros, M. Barranco, D. Gessner, S. Djerasevic, J. Proenza, Fault tolerance in highly reliable ethernet-based industrial systems, Proceedings of the IEEE 107 (6) (June 2019) 977–1010.

[2] A. Avizienis, J.C. Laprie, B. Randell, C. Landwehr, Basic concepts and taxonomy of dependable and secure computing, IEEE Transactions on Dependable and Secure Computing 1 (1) (October 2004) 11–33.

[3] B. Baudry, M. Monperrus, The multiple facets of software diversity: recent developments in the year 2000 and beyond, ACM Computing Surveys 48 (1) (September 2015) 16.

[4] Dependable Systems and Networks (DSN) Conference, http://www.dsn.org.

[5] E. Dubrova, Fault-Tolerant Design, Springer, 2013.

[6] W.R. Dunn, Practical Design of Safety-Critical Computer Systems, Reliability Press, 2002.

[7] C.E. Ebeling, An Introduction to Reliability and Maintainability Engineering, McGraw-Hill, 1997.

[8] C. Edwards, T. Lombaerts, H. Smaili, Fault-Tolerant Flight Control, Springer, 2009.

[9] B. Fuhrt, A. Escalante, Handbook of Cloud Computing, Springer, 2010.

[10] J-C. Geffroy, G. Motet, Design of Dependable Computing Systems, Kluwer Academic Publishers, 2002.

[11] R. Ghosh, K.S. Trivedi, V.K. Naik, D.S. Kim, End-to-end performability analysis for infrastructure-as-a-service cloud: an interacting stochastic models approach, in: Pacific Rim International Symposium on Dependable Computing, 2010, pp. 125–132.

[12] R.S. Hammer, Patterns for Fault-Tolerant Software, John Wiley, 2013.

[13] T. Herault, Y. Robert, Fault-Tolerance Techniques for High-Performance Computing, Springer, 2015.

[14] P. Jalote, Fault Tolerance in Distributed Systems, PTR Prentice Hall, 1994.

[15] J. Knight, Fundamentals of Dependable Computing for Software Engineers, Chapman and Hall, 2012.

[16] J.F. Meyer, On evaluating the performability of degradable computing systems, IEEE Transactions on Computers 29 (August 1980) 720–731.

[17] G. Psychou, D. Rodopoulos, M.M. Sabry, D. Atienza, T.G. Noll, F. Catthoor, Classification of resilience techniques against functional errors at higher abstraction layers of digital systems, ACM Computing Surveys 50 (4) (2017) 50.

[18] L.L. Pullum, Software Fault Tolerance Techniques and Implementation, Artech House, 2001.

[19] D.P. Siewiorek, R.S. Swarz, Reliable Computer Systems: Design and Evaluation, A. K. Peters, 1998.

[20] M.L. Shooman, Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design, Wiley-Interscience, 2001.
[21] K.S. Trivedi, Probability and Statistics With Reliability, Queuing, and Computer Science Applications, John Wiley, 2002.
[22] M. Weiser, The computer for the twenty-first century, Scientific American 265 (3) (September 1991) 94–105.

CHAPTER

HARDWARE FAULT TOLERANCE 2

Hardware fault tolerance is the most mature area in fault-tolerant computing. Many hardware fault-tolerance techniques have been developed and used in practice in critical applications, ranging from telephone exchanges to space missions. In the past, the main obstacle to a wide use of hardware fault tolerance was the cost of the extra hardware required. With the continued reduction in the cost of hardware, this is no longer a significant drawback, and the use of hardware fault-tolerance techniques is expected to increase. However, other constraints, notably on power consumption, may continue to restrict the use of massive redundancy in many applications.

The hardware can broadly be divided into a hierarchy of three levels. (One can obviously have a more detailed division into sublevels, but three levels will do for our purposes.) At the top is the system level. This is the “face” that the entity presents to the operating environment. An example is the computer hardware that controls a modern aircraft. At the second level, the system is composed of multiple modules or components. Examples include individual processor cores, memory modules, and the I/O subsystem. Obviously, each of these modules will themselves be composed of submodules. At the bottom of the hierarchy are individual nanometer-sized devices.

This chapter first discusses hardware failure at a high level. Then, we dive to the device level to explain some major hardware failure mechanisms. After this, we return to the system level to consider more complex systems consisting of multiple components, describe various resilient structures (which have been proposed and implemented), and evaluate their reliability and/or availability. Next, we describe hardware fault-tolerance techniques that have been developed specifically for general-purpose processors. Finally, we discuss *malicious* faults, and investigate the amount of redundancy needed for protecting against these.

2.1 THE RATE OF HARDWARE FAILURE

The failure rate of a hardware component depends on its current age, any voltage or physical shocks that it has suffered, the ambient temperature, and the technology. The dependence on age is usually captured by what is known as the *bathtub curve* (see Fig. 2.1). When components are very young, their tendency to fail is high. This is due to the chance that some components with manufacturing defects slipped through manufacturing quality control and were released. As time goes on, these components are weeded out, and the component spends the bulk of its life showing a fairly constant failure rate (a precise definition of failure rate is presented a little later, but for now an intuitive understanding is sufficient). As it becomes very old, aging effects start to take over, and the failure rate rises again.

Fault-Tolerant Systems. https://doi.org/10.1016/B978-0-12-818105-8.00012-7