1

Definitions, Concepts, and Principles

Francis Tam
Nokia Research Center, Helsinki, Finland

1.1 Introduction

As our society increasingly depends on computer-based systems, the need for making sure that services are provided to end-users continuously has become more urgent. In order to build such a computer system upon which people can depend, a system designer must first of all have a clear idea of all the potential causes that may bring down a system. One should have an understanding of the possible solutions to counter the causes of a system failure. In particular, the costs of candidate solutions in terms of their resource requirements must also be known. Finally, the limits of the eventual system solution that is put in place must be well understood.

Dependability can be defined as the quality of service provided by a system. This definition encompasses different concepts, such as reliability and availability, as attributes of the service provided by a system. Each of these attributes can therefore be used to quantify aspects of the dependability of the overall system. For example, reliability is a measure of the time to failure from an initial reference instant, whereas availability is the probability of obtaining a service at an instant of time. Complex computer systems such as those deployed in telecommunications infrastructure today require a high level of availability, typically 99.999% (five nines) of the time, which amounts to just over five minutes of downtime over a year of continuous operation. This poses a significant challenge for those who need to develop an already complex system with the added expectation that services must be available even in the presence of some failures in the underlying system.

In this chapter, we focus on the definitions, concepts, principles, and means to achieving service availability. We also explain all the conceptual underpinning needed by the readers in understanding the remaining parts of this book.
1.2 Why Service Availability?

In this section, we examine why the study on service availability is important. It begins with a dossier on unavailability of services and discusses the consequences when the expected services are not available. The issues and challenges related to service availability are then introduced.

1.2.1 Dossier on Unavailability of Service

Service availability – what is it? Before we delve into all the details, perhaps we could step back and ask why service availability is important. The answer lies readily from the consequences when the desired services are not available. A dossier on the unavailability of services aims to illustrate this point.

Imagine you were one of the one million mobile phone users in Finland, who was affected by a widespread disturbance of a mobile telephone service [1] and had problems receiving your incoming calls and text messages. The interrupt of service, reportedly caused by a data overload in the network, lasted for about seven hours during the day. You could also picture yourself as one of the four million mobile phone subscribers in Sweden when a fault, although not specified, had caused the network to fail and unable to provide you with mobile phones services [2]. The disruption lasted for about twelve hours, which began in the afternoon and continued until around midnight.

Although the reported number of people affected in both cases does not seem to be that high at first glance, one has to put them in the context of their populations. The two countries have respectively 5 and 9 millions of people so the proportion of the affected were considerable.

These two examples have given a somewhat narrow illustration of the consequences when services are unavailable in the mobile communication domain. There are many others and they touch on different kinds of services, and therefore different consequences as a result. One case in point was the financial sector reported that a software glitch, apparently caused by a new system upgrade, had resulted in a 5.5 hour delay in shares trading across the Nordic region including Stockholm, Copenhagen, Helsinki, as well as the Baltic and Icelandic stock exchanges [3]. The consequence was significantly high in terms of the projected financial loss due to the delayed opening of the stock market trading.

Another high-profile and high-impact computer system failure was at the Amazon Web Services [4] for providing web hosting services by means of its cloud infrastructure to many web sites. The failure was reportedly caused by an upgrade of network capacity and lasted for almost four days before the last affected consumer data were recovered [5], although 0.07% of the affected data could not be restored. The consequence of this failure was the unavailability of services to the end customers of the web sites using the hosting services. Amazon had also paid 10-day service credits to those affected customers.

A nonexhaustive list of failures and downtime incidents collected by researchers [6] gives further examples of causes and consequences, which includes categories of data center failures, upgrade-related failures, e-commerce system failures, and mission-critical system failures. Practitioners in the field also maintain a list of service outage examples [7]. These descriptions further demonstrate the relationships between the cause and consequence of failures to providing services. Although some of the causes may be of a similar nature to have made the service unavailable in the first place, the consequences are very much dependent on what the computer system is used for. As described in the list of failure incidents, this could range from the inconvenience of not having the service immediately available, financial loss, to the most serious result of endangering human lives.

It is important to note that all the consequences in the dossier above are viewed from the end-users’ perspective, for example, mobile phone users, stockbrokers trading in the financial market and users of web site hosting services. Service availability is measured by an end-user in order to gauge the level of a provided service in terms of the proportion of time it is operational and ready to deliver. This is a user experience of how ready the provided service is. Service availability is a product of the availability of all the elements involved in delivering the service. In the example case of a mobile
phone user above, the elements include all the underlying hardware, software, and networks of the mobile network infrastructure.

1.2.2  Issues and Challenges

Lack of a common terminology and complexity have been identified as the issues and challenges related to service availability. They are introduced in this section.

1.2.2.1  Lack of a Common Terminology

Studies on dependability have long been carried out by the hardware as well as software communities. Because of the different characteristics and as a result a different perspective on the subject, dissimilar terminologies have been developed independently by many groups. The infamous observation of ‘one man’s error is another man’s fault’ is often cited as an example of confusing and sometimes contradictory terms used in the dependability community. The IFIP (International Federation for Information Processing) Working Group WG10.4 on Dependable Computing and Fault Tolerance [8] has long been working on unifying the concepts and terminologies used in the dependability community. The first taxonomy of dependability concepts and terms was published in 1985 [9]. Since then, a revised version was published in [10]. This taxonomy is widely used and referenced by researchers, practitioners, and the like in the field. In this book, we adopt this conceptual framework by following the defined concepts and terms in the taxonomy. On the general computing side, where appropriate, we also use the Institute of Electrical and Electronics Engineers (IEEE) standard glossary of software engineering terminology [11]. The remainder of this chapter presents all the needed definitions, concepts, and principles for a reader to understand the remaining parts of the book.

1.2.2.2  Complexity and Large-Scale Development

Dependable systems are inherently complex. The issues to be dealt with are usually closely intertwined because they have to deal with the normal functional requirements as well as the nonfunctional requirements such as service availability within a single system. Also, these systems tend to be large, such as mobile phone or cloud computing infrastructures as discussed in the earlier examples. The challenge is to manage the sheer scale of development and at the same time, ensure that the delivered service is available at an acceptable level most of the time. On the other hand, there is clearly a common element of service availability implementation across all these wide-ranging application systems. If we can extract the essence of service availability and turn it into some form of general application support, it can then be reused as ready-made template for service availability components. The principle behind this idea is not new. Over almost two decades ago, the use of commercial-off-the-shelf (COTS) components had been advocated as a way of reducing development and maintenance costs by buying instead of building everything from scratch. Since then, many government and business programs have mandated the use of COTS. For example, the United States Department of Defense has included this term into the Federal Acquisition Regulation (FAR) [12].

Following a similar consideration in [13] to combine the complementary notions of COTS and open systems, the Service Availability Forum was established and it developed the first open standards on service availability. Open standards is an important vehicle to ensure that different parts are working together in an ecosystem through well-defined interfaces. The additional benefit of open standards is the reduction of risks in a vendor lock-in for supplying COTS. In the next chapter, the background and motivations behind the creation of the Service Availability Forum and the service availability standards are described. A thorough discussion on the standards’ services and frameworks, including
the application programming and system administrator and management interfaces, are contained in Part Two of the book.

1.3 Service Availability Fundamentals

This section explains the basic definitions, concepts, and principles involving service availability without going into a specific type of computer system. This is deemed appropriate as the consequences of system failures are application dependent; it is therefore important to understand the fundamentals instead of going into every conceivable scenario. The section provides definitions of system, behavior, and service. It gives an overview of the dependable computing taxonomy and discusses the appropriate concepts.

1.3.1 System, Behavior, and Service

A system can be generically viewed as an entity that intends to perform some functions. Such entity interacts with other systems, which may be hardware, software, or the physical world. Relative to a given system, the other entities with which it interacts are considered as its environment. The system boundary defines the limit of a system and marks the place where the system and its environment interact.

Figure 1.1 shows the interaction between a given system and its environment over the system boundary. A system is structurally composed of a set of components bound together. Each component is another system and this recursive definition stops when a component is regarded as atomic, where further decomposition is not of interest. For the sake of simplicity, the remaining discussions in this chapter related to the properties, characteristics, and design approaches of a system are applicable to a component as well.

The functions of a system are what the system intends to do. They are described in a specification, together with other properties such as the specific qualities (for example, performance) that these

![Figure 1.1 System interaction.](image-url)
functions are expected to deliver. What the system does to implement these functions is regarded as its behavior. It is represented by a sequence of states, some of which are internal to the system while some others are externally visible from other systems over the system boundary.

The service provided by a system is the observed behavior at the system boundary between the providing system and its environment. This means that a service user sees a sequence of the provider’s external states. A correct service is delivered when the observed behavior matches those of the corresponding function as described in the specification. A service failure is said to have occurred when the observed behavior deviates from those of the corresponding function as stated in the specification, resulting in the system delivering an incorrect service. Figure 1.2 presents the transition from a correct service to service failure and vice versa. The duration of a system delivering an incorrect service is known as a service outage. After the restoration of the incorrect service, the system continues to provide a correct service.

Take a car as an example system. At the highest level, it is an entity to provide a transport service. It primarily interacts with the driver in its environment. A car system is composed of many smaller components: engine, body, tires, to name just a few. An engine can be further broken into smaller components such as cylinders, spark plugs, valves, pistons, and so on. Each of these smaller components is connected and interacts with other components of systems.

As an example, an automatic climate control system provides the drivers with a service to maintain a user-selected interior temperature inside the car. This service is usually implemented by picking the proper combination of air conditioning, heating, and ventilation in order to keep the interior temperature at the same level. The climate control system must therefore have functions to detect the current temperature, turn on or off the heater and air conditioning, and open or close air vents. These functions are described in the functional specification of the climate control system, with clear specifications of other properties such as performance and operating conditions.

Assuming that the current interior temperature is 18°C and the user-selected temperature is 20°C, the expected behavior of the automatic climate control system is to find out the current temperature and then turn on the heater until the desired temperature is reached. During these steps, the system goes through a sequence of states in order to achieve its goal. However, not all the states are visible to the driver. For example, the state of the automatic climate control system with which the heater interacts is a matter of implementation. Indeed whether the system uses the heater or air conditioning to reach the user-selected temperature is of no interest to the user. On the other hand, the state showing the current interior temperature is of interest to a user. This gives some assurance that the temperature is changing in the right direction. This generally offers the confidence that the system is providing the correct service. If for some reason the heater component breaks down, the same sequence of steps does not raise the interior temperature to the desired 20°C as a result. In this case, the system has a service failure because the observed behavior differs from the specified function of maintaining a user-selected temperature in the car. The service outage can be thought of as the period of time when the heater breaks down until it is repaired, possibly in a garage by qualified personnel and potentially takes days.
1.3.2 Dependable Computing Concepts

As discussed in the introduction, availability is one part of the bigger dependability concept. The term dependability has long been regarded as an integrating concept covering the qualities of a system such as availability, reliability, safety, integrity, and maintainability. A widely agreed definition of dependability [10] is ‘the ability to deliver service that can justifiably be trusted.’ The alternative definition, ‘the ability to avoid service failures that are more frequent and severe than is acceptable’ is very often served as a criterion to decide if a system is dependable or not.

Figure 1.3 shows the organization of the classifications. At the heart is the main concept of dependability, which is comprised of three subconcepts: threats, attributes, and means. It must be pointed out that the concept of security has been taken out due to the subject being outside the scope of this book. A threat is a kind of impairment that can prevent a system from delivering the intended service to a user. Failures, errors, and faults are the kinds of threats that can be found in a system. Since dependability is an integrating concept, it includes various qualities that are known as attributes. These include availability, reliability, safety, integrity, and maintainability of the intended service. The means are the ways of achieving the dependability goal of a service. To this end, four major groups of methods have been developed over the years, namely, fault prevention, fault tolerance, fault removal, and fault forecasting.

1.3.2.1 Threats

In order to understand the consequences of a threat to a service, it is important to differentiate the different types of threats and their relationship. The fault–error–failure model expresses that a fault, a physical defect found in a system, causes an error to the internal state of a system, and in turn finally causes a failure to a system, which can be detected externally by users. Faults are physical defects and that means they could be wiring problems, aging of components, and in software an incorrect design. The existence of a fault does not mean that it immediately causes an error and then a failure. This is because the part of the system that is affected by the fault may not be running all the time. A fault is said to be in a dormant state until it becomes active when the part of the system affected is exercised.
The activation of a fault brings about an error, which is a deviation from the correct behavior as described in the specification. Since a system is made up of a set of interacting components, a failure does not occur as long as the error caused by a fault in the component’s service state is not part of the external service state of the system.

### 1.3.2.2 Attributes

- **Reliability**
  This is defined as the ability of a system to perform a specified function correctly under the stated conditions for a defined period of time.

- **Availability**
  This is defined as the proportion of time when a system is in a condition that is ready to perform the specified functions.

- **Safety**
  This is defined as the absence of the risk of endangering human lives and of causing catastrophic consequences to the environment.

- **Integrity**
  This is defined as the absence of unauthorized and incorrect system modifications to its data and system states.

- **Maintainability**
  This is defined as a measure of how easy it is for a system to undergo modifications after its delivery in order to correct faults, prevent problems from causing system failure, improve performance, or adapt to a changed environment.

### 1.3.2.3 Means

- **Fault prevention**
  This is defined as ensuring that an implemented system does not contain any faults. The aim is to avoid or reduce the likelihood of introducing faults into a system in the first place. Various fault prevention techniques are usually carried out at different stages of the development process. Using an example from software development, the use of formal methods in the specification stage helps avoid incomplete or ambiguous specifications. By using well-established practices such as information hiding and strongly typed programming languages, the chances of introducing faults in the design stage are reduced. During the production stage, different types of quality control are employed to verify that the final product is up to the expected standard. In short, these are the accepted good practices of software engineering used in software development. It is important to note that in spite of using fault prevention, faults may still be introduced into a system. Therefore, it does not guarantee a failure-free system. When such a fault activates during operational time, this may cause a system failure.

- **Fault tolerance**
  This is defined as enabling a system to continue its normal operation in the presence of faults. Very often, this is carried out without any human intervention. The approach consists of the error detection and system recovery phases. Error detection is about identifying the situation where the internal state of a system is different from that of a correct one. By using either error handling or fault handling in the recovery phase, a system can perform correct operations from this point onwards. Error handling changes a system state that contains errors into a state without any detected errors. In this case, this action does not necessarily correct the fault that causes the errors. On the other hand, a system using fault handling in the recovery phase essentially repairs the fault that causes the errors. The workings of fault tolerance are presented in Section 1.4 in more details.
• **Fault removal**
  This achieves the dependability goal by following the three steps of verification, diagnosis, and correction. Removal of a fault can be carried out during development time or operational time. During the development phase, this could be done by validating the specification; verifying the implementation by analyzing the system, or exercising the system through testing. During the operational phase, fault removal is typically carried out as part of maintenance, which first of all isolates the fault before removing it. Corrective maintenance removes reported faults while preventive maintenance attempts to uncover dormant faults and then removes them afterwards. In general, maintenance is a manual operation and it is likely to be performed while the system is taken out of service. A fault-tolerant system, on the other hand, may be able to remove a fault without disrupting service delivery.

• **Fault forecasting**
  This is concerned with evaluating the system behavior against likely failures. This involves identifying and classifying the failure modes and assessing how well a system deals with anticipated faults in terms of probabilities. The assessment is usually carried out by two complementary quantitative evaluation approaches: modeling and operational testing. A behavioral model of the system is first created and then processed to estimate the dependability measure of the system. The data used in processing the model can either be obtained by operational testing, which is performed in the system’s operational environment, or based on data from past system failures. It is important to point out that fault forecasting only provides a prediction of how well the current system copes with anticipated faults. If appropriate actions are taken, for example, by using fault removal, the quality of the system would improve over time. However, there are situations when the identified faults are not removed from the system. This is usually due to economic reasons such as the costs of removing a fault outweighing its benefit, especially when the probability of the fault occurring is low. If a fault is elusive and difficult to pin down, then fault tolerance could be an alternative to fault removal.

It would be naïve to believe that any one of the four approaches can be used on its own to develop systems with high dependability. In practice, all or a combination of the methods are used at each design and implementation stage of developing dependable systems. For example, it is common to have extensive fault prevention and fault removals throughout the various system development stages in a fault-tolerant system. After all, one would not want any faults to be introduced into the critical phases of error detection and system recovery. Once a system is operational, live data are used in fault forecasting in order to feed in the improvements and/or corrections for the next version of the system.

### 1.3.3 The Meaning of Availability

Due to the fact that studies of dependable systems have been carried out by a diverse group of communities, there is a constant source of confusion over the use of some terms. Specifically, some definitions associated with the measures of failures must be explained. Before we discuss the meaning of availability, we first give the following definitions and then discuss their relationships in the context of expressing a system’s availability.

• **Mean time to failure (MTTF).** This is defined as the expected time that a system takes to fail. It is the same as uptime, which can be described as the duration of time when a system is operational and providing its service.

• **Mean time to repair (MTTR).** This is defined as the expected time required to repair a system such that it can return to its normal, correct operation. Assuming that all the overheads are accounted for in arriving at the repair time estimate, it should be the same as the downtime. MTTR is known
to be difficult to estimate. One possible approach is to inject faults into a system and then determine experimentally by measuring the time to fix them.

- **Mean time between failures (MTBF).** This is defined as the expected time between two successive failures in a system. This measure is meaningful only if a system can be repaired, that is, a repairable system, because a system fails the second time only if it has been repaired after the first failure.

Figure 1.4 illustrates the relationship among the different failure measures in a system. It shows the status of a system, which can be either up or down, against the time of operation. At time $T_{\text{start}}$, the system starts its operation and continues until time $T_{\text{failure-1}}$, when it encounters the first failure. After the duration of MTTR, which is the downtime, the system is repaired and continues its normal operation at time $T_{\text{repaired-1}}$. It continues until the second failure hits the system at time $T_{\text{failure-2}}$.

As shown in Figure 1.4, the relationship among the failure measures of a system can be expressed as:

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

Another way of expressing the MTBF of a system is the sum of the downtime and uptime during that period of time. This is essentially linked to a system’s dependability attribute of availability.

Availability is defined as the degree to which a system is functioning and is accessible to deliver its services during a given time interval. It is often expressed as a probability representing the proportion of the time when a system is in a condition that is ready to perform the specified functions. Note that the time interval in question must also include the time of repairing the system after a failure. As a result, the measure of availability can be expressed as:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTBF}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

The availability measure is presented as a percentage of time a system is able to provide its services readily during an interval of time. For example, an availability of 100% means that a system has no downtime at all. This expression also highlights the fact that the availability of a system depends on how frequently it is down and how quickly it can be repaired.

It is generally accepted that a highly available system, such as those used in telecommunications, must have at least 99.999% availability, the so-called systems with 5–9s availability requirements. Table 1.1 shows the maximum allowable downtime of a system against the different number of 9s availability required under various operating intervals. As shown, a 5–9s system allows for just 5 minutes 15 seconds of downtime in 1 year’s continuous operation.

![Figure 1.4](image)
Table 1.1  Maximum allowable downtime for different availability levels

<table>
<thead>
<tr>
<th>Availability</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Years of continuous operations</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>99.0000% (2–9s)</td>
<td>3 d 15 h 36 min 0 s</td>
<td>7 d 7 h 12 min 0 s</td>
<td>10 d 22 h 48 min 0 s</td>
</tr>
<tr>
<td>99.9000% (3–9s)</td>
<td>8 h 45 min 15 s</td>
<td>17 h 31 min 12 s</td>
<td>1 d 2 h 16 min 48 s</td>
</tr>
<tr>
<td>99.9900% (4–9s)</td>
<td>52 min 34 s</td>
<td>1 h 45 min 7 s</td>
<td>2 h 37 min 41 s</td>
</tr>
<tr>
<td>99.9990% (5–9s)</td>
<td>5 min 15 s</td>
<td>10 min 31 s</td>
<td>15 min 46 s</td>
</tr>
<tr>
<td>99.9999% (6–9s)</td>
<td>32 s</td>
<td>1 min 3 s</td>
<td>1 min 35 s</td>
</tr>
</tbody>
</table>

In order to relate to the service users of a system, the availability expression above can be rewritten for a service in terms of service uptime and service outage as:

\[
\text{Service availability} = \frac{\text{service uptime}}{\text{service uptime} + \text{service outage}}
\]

As stated in the definition in Section 1.2, a measure of service availability is the product of the availability of all the elements involved in implementing the service. This turns out to be more complicated than just looking at the plainly stated downtime for a single element. In reality, a system implementing end-user services is typically composed of many subsystems. The availability calculation must therefore take into account the availability of all the constituent subsystems. In general, subsystems must achieve a high level of availability in order to meet the expected service availability requirement. In practice, however, the availability requirements for each subsystem are more fine-grained and are usually attached to individual functions, instead of a single figure for the entire subsystem. Nevertheless, a whole range of possible failures such as hardware, operating system, application, operator errors, and environmental problems may still contribute to the piling up of downtimes in the availability calculation equation.

While the main source of impacting service availability has been the failure of some of the underlying elements, the need to modify a system after it has been put into operation is increasingly becoming a vital factor for system designers to consider. All systems need changes, be it an upgrade or replacement of hardware components, new features or simply bug fixes. In order to deal with an increase in service capacity, for instance, more service requests per second, additional hardware is typically put into the system to increase its storage and/or processing capability. In the case of repairing a faulty hardware component, it is normally taken away and replaced by a new and functional one. Other required changes at some regular intervals include operating system changes and new application versions.

The traditional way of dealing with changes or upgrades is to stop the system and carry out the modifications accordingly. This obviously makes an impact on the service availability measure because from the end-users’ perspective, the service is no longer operational to offer the intended function. This is also deemed unacceptable to a service operator because the service outage is translated into loss of revenues. A mobile phone service is a case in point. In addition, upgrading an operational system has been known to be error-prone [3, 5] primarily due to its intrinsic complexity. Therefore, some methods must be devised to keep the services going while an upgrade is carried out on the operational system. As the society is depending on more and more computer-based services nowadays, there is a clear trend to expect services to be available in more and more systems even during their upgrades. Even if service unavailability is inevitable, the disruption must be kept to a minimum to reduce its impact on users. This is especially true for those systems that have a long mission time and this requirement cannot be underestimated.

In the next section, we will discuss how service availability can be achieved for both unplanned and planned events, primarily caused by failures and upgrades respectively.
1.4 Achieving Service Availability

This section focuses on how the notion of service availability can be realized in systems. It shows how fault tolerance can be used as the basis for delivering service availability to systems. It explains various forms of protective redundancy and the workings of fault tolerance. It also highlights the interrelationship between upgrade without loss of service and protective redundancy in the context of providing service availability in a system.

1.4.1 Following the Framework of Fault Tolerance

Fault tolerance aims to be a failure avoidance approach. It attempts to intervene and defend against faults from causing failures. The general assumption is that in spite of all the fault prevention methods employed, there is still a chance that faults can creep into a system and cause a failure. The basic principle behind fault tolerance is to ensure that a system is sufficiently designed to withstand such possibility by means of error detection and system recovery. The approach tries to make sure that if a fault is activated, the caused error can be detected and then handled accordingly before it causes a failure.

Fault tolerance is considered to be equally applicable to implementing service availability for unplanned events such as those caused by failures of the underlying system. The four phases of fault tolerance [14] are outlined below to show how the principles can be applied to delivering service availability:

1. **Error detection.**
   Before any action can be taken to address a service failure, the presence of an error must first be identified.

2. **Damage confinement and assessment.**
   The level of damage caused by a fault is evaluated and if possible, the effect of the error should be restrained as much as possible. It is worth pointing out that an error can be passed as data, which may be in the form of either user data over the service interface or as system state information. This phase basically tries to limit the scope and propagation of an error from one part of the system to another.

3. **Error recovery.**
   Error recovery is the process of transforming an erroneous system state into one that has no detected errors. Therefore, the possibility of the activated fault causing a service failure can be eliminated.

4. **Fault treatment and service continuation.**
   If the fault that causes the detected error to occur can be identified and corrected, it could prevent the same fault from being activated again. Even without this fault handling, a system can continue to provide its intended service because the error condition has now been cleared. The key concern in this phase is the attempt to create a perception to service users that the intended service continues to be available as if nothing has happened.

In the ideal case, a service continues operating regardless of what kinds of faults it encounters. Due to the application requirements and/or resource constraints, a service implementation does not necessarily have all of the above fault tolerance phases. As always, there is a trade-off between the resources needed and the protection it offers. There are also the influences of the application requirements. For example, some application scenarios have strict requirements on the types of faults to be tolerated, while other application scenarios have less stringent requirements. Some applications may even allow for service failures provided that certain safety conditions are fulfilled. Some applications are in between, where a partial service failure is acceptable.

In a traditional fault-tolerant system, the failure response defines how such a system reacts when a failure is encountered. As a result, the response can be viewed as the level of fault tolerance provided by a system. The same definitions are also relevant to services:
• **Fail-operational**
  A service continues to operate in the presence of errors with no loss of functionality or performance. This is the highest level of service availability as the users do not even notice any significant differences in terms of their expectations. Needless to say, the required additional resources to cover all the conceivable failure scenarios are usually prohibitively high for most applications.

• **Fail-soft (graceful degradation)**
  A service continues to operate in the presence of errors with a degradation of functionality and/or performance. This is perhaps the most common level of application requirements. The idea behind this is to at least keep some of the functions going with the current, limited resources available. The system can subsequently be taken off for repair before full service is restored. The choices of which functions are kept naturally depend upon the types of error the system is facing. They also depend on how an application is designed. The most intuitive line of thinking would be placing more protection on critical functions than their less essential counterparts. Relaxing nonfunctional requirements such as performance at the time the system is in trouble is another option available to a system designer. For example, reducing serving capacity would put less strain on the demand of limited resources.

• **Fail-safe**
  A system maintains its integrity and halts its intended operation. It must be pointed out that halting the intended operation does not necessarily mean stopping the system. For example, a traffic light reverting to flashing reds in all directions [11] is not the intended operation for directing traffic. Instead, it is a warning signal to all road users that there is something wrong with the traffic lights. This is a safety measure to help ensure that the possibility of lights erroneously going into greens in all directions when there is a failure is excluded. The primary concern in this type of application is safety. However, the precise meaning of what is a safe condition is entirely application dependent: a service being stopped and not doing anything is a possible and sometimes applicable scenario.

• **Fail-stop**
  A service halts immediately upon a detected malfunction. In some literature, this property is also a synonym for *fail-fast*, which adds the connotation that the stopping is done very quickly. It is a useful and desirable property if a system cooperates with other systems, especially when it prevents the erroneous system from spreading incorrect information to others.

1.4.2 **Redundancy is a Requisite**

The key to a fault tolerance approach is to have protective redundancy built into a system. Protective redundancy is an additional resource that would be unnecessary if a system operates correctly. That is why the term redundant. Protective redundancy, or redundancy for short, is usually a replica of a resource. In the event that a fault causes an error to occur, the replicated resource is used instead of the erroneous one. Thus, the replica protects a system against failures, giving the impression that services are uninterrupted even when there are failures in the underlying system. It is precisely this property that makes fault tolerance stand out as a candidate approach (discussed in Section 1.3.2) that can be used as a basis for achieving service availability. There are two main aspects related to redundancy that require consideration: what should be replicated and how the redundancy is structured in order to achieve the desired service availability.

There are many forms of resources that can be replicated in order to provide redundancy in a system, for example, hardware, software, communications, information, and even time. Duplicating a piece of hardware is perhaps the most common form of redundancy to deal with hardware faults. Running a copy of a piece of software on different hardware can tolerate hardware faults. It must be stressed that if the fault lies in the software, which is basically a design fault, a copy of the same software in the exact same state with the exact same data would only duplicate the same fault. A whole range of solutions based on diversity can be found in the additional reading [15]. Having
more than one communications path among interconnected systems is a way to handle failures in connectivity. Maintaining multiple copies of critical information or data has long been used as a mechanism for dealing with failures in general. This inevitably brings in the issues of consistency among replicas. Simply by repeating the execution of a task has been known to be able to correct transient hardware faults. This is also referred to as temporal redundancy as time is the additional resource in this context. In practice, a combination of these forms of redundancy is used according to the application requirements and resources available.

Since there are replicated resources in the system, it is necessary to differentiate what roles each of these resources should take. A resource is considered to have an active role if it is used primarily to deliver the intended service. The redundant resource is regarded as taking a standby role if it is ready to take over the active role and continue delivering the service when the current active element fails. It must be noted that a standby element usually needs to follow what the current active element is up to in order to be successful in taking over. This typically requires that a standby element has up-to-date state information of the current active element, implying that there are communications between active and standby elements. The end result is that users are given the perception that a service continues as if there was no failure.

The most basic structure for a system employing protective redundancy is to have one active element handling all the service requests, while a standby element tracks the current state of its active counterpart. If the active element fails, the standby element takes over the active role and continues to deliver the intended service. The manner in which resources are structured in terms of their roles in a system is known as the redundancy model.

Figure 1.5 illustrates a frequently used redundancy model known as active-standby involving both the active and standby roles. As shown in (a), an active role is taken to provide service S by an application A running on computer node X under normal operation. The application is replicated on node Y and assumes the standby role of providing service S. During this time, the necessary information for the standby application to take over the active role must be obtained and maintained. If there is a failure on node X rendering the application in the active role to fail as well, as depicted in (b), the application with the standby role takes over and continues to provide service S. At this point, however, there is no redundancy to protect further failures. If the failed node X is not repaired, a further failure on node Y will cause service outage.

There are many ways of structuring redundancy in a system, involving different numbers of active or standby roles, each of which has resource utilization and response time implications. For example, it is possible to have one standby element protecting a number of active elements in order to increase the overall resource utilization. With only a single redundant element, the resulting system will not be protected against further failures if the faulty element is not repaired. The trade-off of how many active and standby elements and in what way they are structured must be weighed against the application

![Figure 1.5](image-url)
requirements. It is worth noting that with the coordination of the active and standby roles, an application becomes distributed in nature and that makes it more complex. In Chapter 6, some of the commonly used redundancy models in the context of managing service availability are discussed in more details.

1.4.3 Dealing with Failures

Fault tolerance is performed by means of error detection and then system recovery. In this section, we look at the mechanics of fault tolerance in more detail during the four phases, in particular, how service availability can be accomplished by following the framework of a fault tolerance approach.

1.4.3.1 Error Detection

Error detection aims to identify the occurrence when the state in a system deviates from its expected behavior as specified. This can be done by either the system itself or an external entity monitoring the system. There are many ways to detect an error and the precise mechanism used is obviously application dependent. In general, these techniques fall into two main types: concurrent detection and pre-emptive detection. The main difference is that concurrent detection is carried out during normal service delivery while pre-emptive detection is performed when normal service delivery is suspended. Pre-emptive detection aims at finding latent errors and dormant faults. Although the service is unavailable in the case of the pre-emptive detection type, it is worth mentioning that its use may still be appropriate for some applications provided that the anticipated service outage is within the allowable limit.

1.4.3.2 Damage Confinement and Assessment

The main purpose of damage confinement and assessment is to contain the detected error and limit its effect from spreading to other parts of a system. At design time, a commonly used approach is to hierarchically decompose a system into smaller and manageable parts. Not only is each part a unit of service, but also a unit of failure [16]. If such a unit fails, it is replaced by a similar, functional unit. The notion of a field replaceable unit (FRU) has long been used in hardware. The equivalent in software is usually a software module. However, defining a unit of failure for software is more complicated because software tends to be more complex due to its structure, and that different pieces of software usually cooperate to achieve some functions. As pointed out in a previous section, errors can be spread via incorrect data among these cooperating software entities. Therefore, the boundary of a unit of failure changes depending on what the fault is.

A fault zone is used for fault isolation and repair. It defines the boundary of a unit of failure for a specific fault and can be viewed as a collection of entities that may be damaged by this fault. This can therefore be used to determine how widespread the effect of this fault may have when it is activated. A fault zone is associated with a recovery strategy so that appropriate corrective actions can be taken in the recovery phase.

Figure 1.6 shows the structure of a simple application and is used to illustrate the concept of a fault zone in isolating faults. The application, which is composed of three subsystems, interacts with its environment. Each subsystem, which consists of a number of software modules, has some defined functions to perform. In the communication subsystem, for example, modules A and B cooperate in order to provide the subsystem’s functions. Altogether, the subsystems cooperate to provide the application’s functions. If a fault originates from module A and does not cause any error outside the module, then the fault zone is module A. If a fault originates from within the communication subsystem and may cause errors within the subsystem but not outside, the fault zone is the communication subsystem. If a fault may cause errors all over the application, the corresponding fault zone is the application. The corresponding recovery action of a fault zone will be described in the next section on error recovery under recovery escalation.
1.4.3.3 Error Recovery

Error recovery is essentially a way of correcting the error condition, which results in masking the effect of an error. After a successful recovery action, a system continues to deliver a seemingly error-free service. In general, error recovery does not normally repair the fault that caused the error in the first place. In some cases though, it has the combined effect of error recovery and fault repair. There are many techniques available and they are not mutually exclusive. That is, if an error persists, another method can be attempted next according to some predefined recovery strategy for a system.

- Rollback
  Rollback, or backward error recovery, attempts to restore the system to a correct or error-free state so that it can continue to provide the intended service from this point onwards. It assumes that the state just before an error appears is correct. A rollback mechanism requires the process known as checkpointing to record prior system states at various predetermined recovery points. When an error is detected, the system state is restored to the last saved, presumably error-free state. Rollback does not require the knowledge of the error it deals with and therefore provides a general recovery scheme. However, it takes additional resources for the checkpointing operation and time redundancy to perform.

- Roll-forward
  Roll-forward, or forward error recovery, tries to move the system into a new and correct state so that it can continue from there to provide the intended service. It requires that a new and correct state be found. Roll-forward is commonly used to gracefully degrade a system by moving the system to a state where reduced capability is delivered to its users. It is generally regarded as being quick but it requires application-specific knowledge. In the context of transaction processing, roll-forward is typically performed by restoring a database from a backup and then applying a separately recorded changes to the database to bring it up to date.

- Failover
  A failover action corrects an error condition by using an element that previously has the standby role assigned instead of the one currently assigned the active role to deliver the service. This change of
role assignments is typically used to replace a failed element that has an active role assigned at the
time when the error condition is detected. It usually requires that a system has protective redundancy
in place and a redundancy model defined. A main objective is to enable a service to continue as
soon as possible by using a standby element instead. It assumes that an active element’s service
state has not been corrupted and therefore neither is that of the standby. Typically the element that
has a detected error is not used any more until it has been repaired. At the point when a repaired
element is reintroduced into the system, an option known as fail-back which reassigns the active
role again can be applied. This is usually used in a system where there is a preferred element for
delivering services under normal operating condition, probably due to the richer resources available
in this element.

- **Restart**
  A restart action clears an error by stopping an element and then starting it again in a controlled
manner. The idea is to bring an element back to a normal state, which is usually found at the
beginning of its operation. This term is used in its most generic form in this context and the concept
encompasses all the variations of this action. The differences stem from the fact that different terms
are traditionally associated with hardware and software. For example, reset is commonly used in
referring to hardware, while restart is typically linked to software. Reboot is generally connected
to a computer which normally means initializing its hardware and reloading the operating system.
There are also differences in terms of how a restart operation is carried out, for example, whether an
element is stopped in an abrupt or orderly manner; how many changes to the element’s settings are
retained when its operation starts again; and if power is taken off from the element or not are just
some of these variations. Caution is therefore required when this term is interpreted with reference
to the precise actions taken.

It must be noted that a recovery action can normally mask an error condition only at the place
where the error was detected. However, if the fault does not originate from the affected element
and it has not been repaired, the same error condition is likely to return, in addition to which more
errors may be detected on other elements due to the spreading of damage caused by the fault. In a
system where fault zone is used for isolating faults, recovery escalation can be used to raise the error
condition to its containing fault zone, that is, the next higher level of fault zone. The recovery action
associated with this fault zone is carried out on all the entities in the same fault zone. If the recovery
action in the new fault zone still cannot remove this persistent error condition, this process continues
to the next higher level until the condition is cleared or there are no more levels to go up to. It is
important to remember that due to the hierarchical structure, the higher the level of fault zone goes,
the more entities are impacted during the corresponding recovery action. If services of the affected
elements are disrupted during a recovery action, this may potentially have an impact on the overall
service availability of the system.

In calculating the service availability measure for a system, recovery time is an important factor.
The smaller the recovery time, the higher the service availability in a system. The recovery times of
various elements in a system, for example, hardware, operating systems, and software must also be
taken into account.

### 1.4.3.4 Fault Treatment and Service Continuation

Fault treatment in this phase aims to stop faults from activating again in a system, thus preventing
faults from causing errors that may lead to a system failure. Treating faults is usually accomplished
by corrective maintenance, which aims to remove faults from a system altogether while a system
is in its operational phase. Handling a fault first of all requires the identification of the fault in
terms of its location and type. This is followed by excluding the faulty elements from taking part
in delivering the service, before a healthy and correctly functioning element is put into the system to replace the faulty one. A new element may be a spare part for hardware, or a new version of program with bug fixes for software. In order to provide service continuation, the new element must be put into the system without stopping it. For hardware, technologies such as hot swap have been successfully used to replace components without the need for taking the power off a system. As for software, it typically requires a restart and therefore it may render services unavailable for a period of time during this operation. However, with protective redundancy deployed in a system to achieve service availability, it is possible to avoid or minimize a service outage if an upgrade is carefully planned.

1.4.4 Upgrade Matters

We have seen in the previous section how unplanned events such as failures are dealt with by following the framework of fault tolerance. In this section, we turn our attention to planned events – the issues of upgrading a system with minimum or no service outage.

In order to reduce the chances of losing availability of a service during an upgrade, an intuitive way is to decrease the scope of affected elements in a system to the smallest possible. In a system where protective redundancy is already put in place for dealing with anticipated failures, it can also be used to keep a service functioning while some parts of the system are being upgraded. Rolling upgrade is one such popular method that attempts to limit the scope of impact by upgrading one part of a system at a time. If the operation is successful, it continues to another part of the system until all the required elements have been upgraded. During this upgrade process, when an element to be upgraded must be taken out of service, the protective redundancy is there to help provide service continuation.

Figure 1.7 shows an example of rolling upgrade and the basic principle behind the method. The objective of the upgrade in the example is to update the currently running service from version 1 (S1) to version 2 (S2). The initial state of the system is shown in (a), where nodes X and Y are configured in an active-standby redundancy mode in which node X assumes the active role whereas node Y is assigned the standby role. When the rolling upgrade begins in (b), the standby role of node Y is taken away when the application is being updated to version 2. Although service S1 is still available at this time, a failure would incur a service outage because the protective redundancy has been removed. Step (c) shows that after a successful upgrade of the application on node Y, it is the turn for node X. At this point, node Y with the newly updated service takes over the active role to provide the service (S2) while node X is taken off for the upgrade. Similar to the condition in step (b), a failure at this point would incur a service outage. Finally in step (d), it shows that node X has been successfully upgraded and therefore can be assigned the standby role, returning the system to its initial state of having an active-standby redundancy model to protect it from service outage.

In practice, there are variations to the basic steps as illustrated. For example, if node X is the preferred provider of the service in an application scenario then it must be reverted back to the active role after a successful upgrade. It must be added that if we want service continuity during an upgrade, S1 needs to behave in the same way as S2. When the system transitions from step (b) to (c), A2 on node Y must synchronize with A1 on node X. Indeed there are many other upgrade methods, including some custom schemes that are needed to meet the requirements of a specific site in terms of its capacity planning and service level agreement. The key issue here is to provide applications with a flexible way for applying different upgrade methods, with the aim of eliminating or reducing the window of vulnerability of service outage (as shown in steps (b) and (c) above), if any.

In case the new version does not work, a plan is also needed to ensure that the system can be restored to the state prior to the upgrade such that the service, albeit the previous version, continues to operate. Therefore, the monitoring and control of the upgrade, together with measures for error recovery of the upgrade process, are essential. Since an upgrade is a planned event, these extra
actions can be taken into consideration and designed into an upgrade operation. A related issue is that error-prone manual processes are typically used in upgrades. Experience has shown that human mistakes account for a considerable number of cases of system crashes during an upgrade, especially in large-scale systems. The upgrade process thus needs to be automated as much as possible. Chapter 9 has an in-depth treatment on the subject of upgrade.

1.5 Conclusion

We have presented the definitions, basic concepts, and principles of dependable computer systems in general, service availability in particular, at the level appropriate for the remaining of the book. By showing the consequences when expected services were not available, we examined why the study on service availability was important. Issues and challenges related to service availability were introduced. We presented how fault tolerance can be used as the basis for delivering service availability to systems. Various forms of protective redundancy have been explained, together with its interrelationship with upgrade without loss of service in the context of delivering service availability to a system.

The subject area of dependable computer systems is a broad one. It is therefore impossible to cover all aspects and the wide-ranging issues in a single chapter. Interested readers are referred to the additional reading list. In [17] an in-depth treatment of designing dependable systems is presented along the development life cycle that includes the specification, design, production, and operation stages. The fundamentals of reliable distributed systems, web services and applications of reliability techniques are thoroughly covered in [18]. Pullum [19] gives an account of the techniques that are based on design and data diversity, which are applicable solutions for tolerating software design faults. A closely related subject of disaster recovery is covered in full in [20]. A document [21] published

---

**Figure 1.7** Rolling upgrade example (a) Initial state: current versions; (b) Application A on node Y is being upgraded; (c) Application A on node X is being upgraded; (d) Upgraded system: new versions.
by the High Availability Forum, a predecessor of the Service Availability Forum, provides an insight into the background, motivation, and the approach to developing the specifications discussed in the rest of this book.

In the next chapter, we will examine the benefits of adopting open standards of service availability in general for carrier-grade systems, and the Service Availability Forum specifications in particular. It includes the rationale behind the development of the Service Availability Forum from both the technical and business perspectives.