5. Availability

With James Scott

*Ninety percent of life is just showing up.*

—Woody Allen

Availability refers to a property of software that it is there and ready to carry out its task when you need it to be. This is a broad perspective and encompasses what is normally called reliability (although it may encompass additional considerations such as downtime due to periodic maintenance). In fact, availability builds upon the concept of reliability by adding the notion of recovery—that is, when the system breaks, it repairs itself. Repair may be accomplished by various means, which we’ll see in this chapter. More precisely, Avižienis and his colleagues have defined dependability:

Dependability is the ability to avoid failures that are more frequent and more severe than is acceptable.

Our definition of availability as an aspect of dependability is this: “Availability refers to the ability of a system to mask or repair faults such that the cumulative service outage period does not exceed a required value over a specified time interval.” These definitions make the concept of failure subject to the judgment of an external agent, possibly a human. They also subsume concepts of reliability, confidentiality, integrity, and any other quality attribute that involves a concept of unacceptable failure.

Availability is closely related to security. A denial-of-service attack is explicitly designed to make a system fail—that is, to make it unavailable. Availability is also closely related to performance, because it may be difficult to tell when a system has failed and when it is simply being outrageously slow to respond. Finally, availability is closely allied with safety, which is concerned with keeping the system from entering a hazardous state and recovering or limiting the damage when it does.

Fundamentally, availability is about minimizing service outage time by mitigating faults. Failure implies visibility to a system or human observer in the environment. That is, a failure is the deviation of the system from its specification, where the deviation is externally visible. One of the most demanding tasks in building a high-availability, fault-tolerant system is to
understand the nature of the failures that can arise during operation (see the sidebar “Planning for Failure”). Once those are understood, mitigation strategies can be designed into the software.

A failure’s cause is called a fault. A fault can be either internal or external to the system under consideration. Intermediate states between the occurrence of a fault and the occurrence of a failure are called errors. Faults can be prevented, tolerated, removed, or forecast. In this way a system becomes “resilient” to faults.

Among the areas with which we are concerned are how system faults are detected, how frequently system faults may occur, what happens when a fault occurs, how long a system is allowed to be out of operation, when faults or failures may occur safely, how faults or failures can be prevented, and what kinds of notifications are required when a failure occurs.

Because a system failure is observable by users, the time to repair is the time until the failure is no longer observable. This may be a brief delay in the response time or it may be the time it takes someone to fly to a remote location in the Andes to repair a piece of mining machinery (as was recounted to us by a person responsible for repairing the software in a mining machine engine). The notion of “observability” can be a tricky one: the Stuxnet virus, as an example, went unobserved for a very long time even though it was doing damage. In addition, we are often concerned with the level of capability that remains when a failure has occurred—a degraded operating mode.

The distinction between faults and failures allows discussion of automatic repair strategies. That is, if code containing a fault is executed but the system is able to recover from the fault without any deviation from specified behavior being observable, there is no failure.

The availability of a system can be calculated as the probability that it will provide the specified services within required bounds over a specified time interval. When referring to hardware, there is a well-known expression used to derive steady-state availability:

\[
\frac{MTBF}{(MTBF + MTTR)}
\]

where \(MTBF\) refers to the mean time between failures and \(MTTR\) refers to the mean time to repair. In the software world, this formula should be interpreted to mean that when thinking about availability, you should think about what will make your system fail, how likely that is to occur, and that there will be some time required to repair it.
From this formula it is possible to calculate probabilities and make claims like “99.999 percent availability,” or a 0.001 percent probability that the system will not be operational when needed. Scheduled downtimes (when the system is intentionally taken out of service) may not be considered when calculating availability, because the system is deemed “not needed” then; of course, this depends on the specific requirements for the system, often encoded in service-level agreements (SLAs). This arrangement may lead to seemingly odd situations where the system is down and users are waiting for it, but the downtime is scheduled and so is not counted against any availability requirements.

In operational systems, faults are detected and correlated prior to being reported and repaired. Fault correlation logic will categorize a fault according to its severity (critical, major, or minor) and service impact (service-affecting or non-service-affecting) in order to provide the system operator with timely and accurate system status and allow for the appropriate repair strategy to be employed. The repair strategy may be automated or may require manual intervention.

The availability provided by a computer system or hosting service is frequently expressed as a service-level agreement. This SLA specifies the availability level that is guaranteed and, usually, the penalties that the computer system or hosting service will suffer if the SLA is violated. The SLA that Amazon provides for its EC2 cloud service is

AWS will use commercially reasonable efforts to make Amazon EC2 available with an Annual Uptime Percentage [defined elsewhere] of at least 99.95% during the Service Year. In the event Amazon EC2 does not meet the Annual Uptime Percentage commitment, you will be eligible to receive a Service Credit as described below.

Table 5.1 provides examples of system availability requirements and associated threshold values for acceptable system downtime, measured over observation periods of 90 days and one year. The term high availability typically refers to designs targeting availability of 99.999 percent (“5 nines”) or greater. By definition or convention, only unscheduled outages contribute to system downtime.

Table 5.1. System Availability Requirements
Planning for Failure

When designing a high-availability or safety-critical system, it’s tempting to say that failure is not an option. It’s a catchy phrase, but it’s a lousy design philosophy. In fact, failure is not only an option, it’s almost inevitable. What will make your system safe and available is planning for the occurrence of failure or (more likely) failures, and handling them with aplomb. The first step is to understand what kinds of failures your system is prone to, and what the consequences of each will be. Here are three well-known techniques for getting a handle on this.

Hazard analysis

_Hazard analysis_ is a technique that attempts to catalog the hazards that can occur during the operation of a system. It categorizes each hazard according to its severity. For example, the DO-178B standard used in the aeronautics industry defines these failure condition levels in terms of their effects on the aircraft, crew, and passengers:

- **Catastrophic.** This kind of failure may cause a crash. This failure represents the loss of critical function required to safely fly and land aircraft.

- **Hazardous.** This kind of failure has a large negative impact on safety or performance, or reduces the ability of the crew to operate the aircraft due to physical distress or a higher workload, or causes serious or fatal injuries among the passengers.

- **Major.** This kind of failure is significant, but has a lesser impact than a Hazardous failure (for example, leads to passenger discomfort rather than injuries) or significantly increases crew workload to the point where safety is affected.

- **Minor.** This kind of failure is noticeable, but has a lesser impact than a Major failure (for example, causing passenger inconvenience or a routine...
flight plan change).
• *No effect.* This kind of failure has no impact on safety, aircraft operation, or crew workload.

Other domains have their own categories and definitions. Hazard analysis also assesses the probability of each hazard occurring. Hazards for which the product of cost and probability exceed some threshold are then made the subject of mitigation activities.

**Fault tree analysis**

*Fault tree analysis* is an analytical technique that specifies a state of the system that negatively impacts safety or reliability, and then analyzes the system’s context and operation to find all the ways that the undesired state could occur. The technique uses a graphic construct (the fault tree) that helps identify all sequential and parallel sequences of contributing faults that will result in the occurrence of the undesired state, which is listed at the top of the tree (the “top event”). The contributing faults might be hardware failures, human errors, software errors, or any other pertinent events that can lead to the undesired state.

**Figure 5.1**, taken from a NASA handbook on fault tree analysis, shows a very simple fault tree for which the top event is failure of component D. It shows that component D can fail if A fails *and* either B or C fails.
Figure 5.1. A simple fault tree. D fails if A fails and either B or C fails.

The symbols that connect the events in a fault tree are called gate symbols, and are taken from Boolean logic diagrams. Figure 5.2 illustrates the notation.

<table>
<thead>
<tr>
<th>Gate Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AND</strong></td>
<td>Output fault occurs if all of the input faults occur</td>
</tr>
<tr>
<td><strong>OR</strong></td>
<td>Output fault occurs if a least one of the input faults occurs</td>
</tr>
<tr>
<td><strong>n</strong></td>
<td>Output fault occurs if n of the input faults occur</td>
</tr>
<tr>
<td><strong>EXCLUSIVE OR</strong></td>
<td>Output fault occurs if exactly one of the input faults occurs</td>
</tr>
<tr>
<td><strong>PRIORITY AND</strong></td>
<td>Output fault occurs if all of the input faults occur in a specific sequence (the sequence is represented by a CONDITIONING EVENT drawn to the right of the gate)</td>
</tr>
<tr>
<td><strong>INHIBIT</strong></td>
<td>Output fault occurs if the (single) input fault occurs in the presence of an enabling condition (the enabling condition is represented by a CONDITIONING EVENT drawn to the right of the gate)</td>
</tr>
</tbody>
</table>

Figure 5.2. Fault tree gate symbols

A fault tree lends itself to static analysis in various ways. For example, a “minimal cut set” is the smallest combination of events along the bottom of the tree that together can cause the top event. The set of minimal cut sets shows all the ways the bottom events can combine to cause the overarching failure. Any singleton minimal cut set reveals a single point of failure, which should be carefully scrutinized. Also, the probabilities of various contributing failures can be combined to come up with a probability of the top event occurring. Dynamic analysis occurs when the order of contributing failures matters. In this case, techniques such as Markov analysis can be used to calculate probability of failure over different failure sequences.

Fault trees aid in system design, but they can also be used to diagnose...
failures at runtime. If the top event has occurred, then (assuming the fault tree model is complete) one or more of the contributing failures has occurred, and the fault tree can be used to track it down and initiate repairs.

Failure Mode, Effects, and Criticality Analysis (FMECA) catalogs the kinds of failures that systems of a given type are prone to, along with how severe the effects of each one can be. FMECA relies on the history of failure of similar systems in the past. Table 5.2, also taken from the NASA handbook, shows the data for a system of redundant amplifiers. Historical data shows that amplifiers fail most often when there is a short circuit or the circuit is left open, but there are several other failure modes as well (lumped together as “Other”).

<table>
<thead>
<tr>
<th>Component</th>
<th>Failure Probability</th>
<th>Failure Mode</th>
<th>% Failures by Mode</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$1 \times 10^{-3}$</td>
<td>Short</td>
<td>5</td>
<td>$X (5 \times 10^{-6})$</td>
</tr>
<tr>
<td>A</td>
<td>$1 \times 10^{-3}$</td>
<td>Other</td>
<td>5</td>
<td>$X (5 \times 10^{-6})$</td>
</tr>
<tr>
<td>B</td>
<td>$1 \times 10^{-3}$</td>
<td>Open</td>
<td>90</td>
<td>$X$</td>
</tr>
<tr>
<td>B</td>
<td>$1 \times 10^{-3}$</td>
<td>Short</td>
<td>5</td>
<td>$X (5 \times 10^{-6})$</td>
</tr>
<tr>
<td>B</td>
<td>$1 \times 10^{-3}$</td>
<td>Other</td>
<td>5</td>
<td>$X (5 \times 10^{-6})$</td>
</tr>
</tbody>
</table>

Adding up the critical column gives us the probability of a critical system failure: $5 \times 10^{-5} + 5 \times 10^{-5} + 5 \times 10^{-5} + 5 \times 10^{-5} = 2 \times 10^{-4}$.

These techniques, and others, are only as good as the knowledge and experience of the people who populate their respective data structures. One of the worst mistakes you can make, according to the NASA handbook, is to let form take priority over substance. That is, don’t let safety engineering become a matter of just filling out the tables. Instead, keep pressing to find out what else can go wrong, and then plan for it.

5.1. Availability General Scenario

From these considerations we can now describe the individual portions of an availability general scenario. These are summarized in Table 5.3:

- *Source of stimulus*. We differentiate between internal and external origins
of faults or failure because the desired system response may be different.

- **Stimulus.** A fault of one of the following classes occurs:
  - **Omission.** A component fails to respond to an input.
  - **Crash.** The component repeatedly suffers omission faults.
  - **Timing.** A component responds but the response is early or late.
  - **Response.** A component responds with an incorrect value.
- **Artifact.** This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.
- **Environment.** The state of the system when the fault or failure occurs may also affect the desired system response. For example, if the system has already seen some faults and is operating in other than normal mode, it may be desirable to shut it down totally. However, if this is the first fault observed, some degradation of response time or function may be preferred.

- **Response.** There are a number of possible reactions to a system fault. First, the fault must be detected and isolated (correlated) before any other response is possible. (One exception to this is when the fault is prevented before it occurs.) After the fault is detected, the system must recover from it. Actions associated with these possibilities include logging the failure, notifying selected users or other systems, taking actions to limit the damage caused by the fault, switching to a degraded mode with either less capacity or less function, shutting down external systems, or becoming unavailable during repair.
- **Response measure.** The response measure can specify an availability percentage, or it can specify a time to detect the fault, time to repair the fault, times or time intervals during which the system must be available, or the duration for which the system must be available.

### Table 5.3. Availability General Scenario
**Figure 5.3** shows a concrete scenario generated from the general scenario: The heartbeat monitor determines that the server is nonresponsive during normal operations. The system informs the operator and continues to operate with no downtime.

<table>
<thead>
<tr>
<th>Portion of Scenario</th>
<th>Possible Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>Internal/external: people, hardware, software, physical infrastructure, physical environment</td>
</tr>
<tr>
<td>Stimulus</td>
<td>Fault: omission, crash, incorrect timing, incorrect response</td>
</tr>
<tr>
<td>Artifact</td>
<td>Processors, communication channels, persistent storage, processes</td>
</tr>
<tr>
<td>Environment</td>
<td>Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation</td>
</tr>
<tr>
<td>Response</td>
<td>Prevent the fault from becoming a failure</td>
</tr>
<tr>
<td></td>
<td>Detect the fault:</td>
</tr>
<tr>
<td></td>
<td>• Log the fault</td>
</tr>
<tr>
<td></td>
<td>• Notify appropriate entities (people or systems)</td>
</tr>
<tr>
<td></td>
<td>Recover from the fault:</td>
</tr>
<tr>
<td></td>
<td>• Disable source of events causing the fault</td>
</tr>
<tr>
<td></td>
<td>• Be temporarily unavailable while repair is being effected</td>
</tr>
<tr>
<td></td>
<td>• Fix or mask the fault/failure or contain the damage it causes</td>
</tr>
<tr>
<td></td>
<td>• Operate in a degraded mode while repair is being effected</td>
</tr>
<tr>
<td>Response Measure</td>
<td>Time or time interval when the system must be available</td>
</tr>
<tr>
<td></td>
<td>Availability percentage (e.g., 99.999%)</td>
</tr>
<tr>
<td></td>
<td>Time to detect the fault</td>
</tr>
<tr>
<td></td>
<td>Time to repair the fault</td>
</tr>
<tr>
<td></td>
<td>Time or time interval in which system can be in degraded mode</td>
</tr>
<tr>
<td></td>
<td>Proportion (e.g., 99%) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing</td>
</tr>
</tbody>
</table>

**Figure 5.3. Sample concrete availability scenario**
5.2. Tactics for Availability

A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system’s actors. A fault (or combination of faults) has the potential to cause a failure. Availability tactics, therefore, are designed to enable a system to endure system faults so that a service being delivered by the system remains compliant with its specification. The tactics we discuss in this section will keep faults from becoming failures or at least bound the effects of the fault and make repair possible. We illustrate this approach in Figure 5.4.

![Figure 5.4. Goal of availability tactics](image)

Availability tactics may be categorized as addressing one of three categories: fault detection, fault recovery, and fault prevention. The tactics categorization for availability is shown in Figure 5.5 (on the next page). Note that it is often the case that these tactics will be provided for you by a software infrastructure, such as a middleware package, so your job as an architect is often one of choosing and assessing (rather than implementing) the right availability tactics and the right combination of tactics.
Detect Faults

Before any system can take action regarding a fault, the presence of the fault must be detected or anticipated. Tactics in this category include the following:

- **Ping/echo** refers to an asynchronous request/response message pair exchanged between nodes, used to determine reachability and the round-trip delay through the associated network path. But the echo also determines that the pinged component is alive and responding correctly. The ping is often sent by a system monitor. Ping/echo requires a time threshold to be set; this threshold tells the pinging component how long to
wait for the echo before considering the pinged component to have failed ("timed out"). Standard implementations of ping/echo are available for nodes interconnected via IP.

• **Monitor.** A monitor is a component that is used to monitor the state of health of various other parts of the system: processors, processes, I/O, memory, and so on. A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack. It orchestrates software using other tactics in this category to detect malfunctioning components. For example, the system monitor can initiate self-tests, or be the component that detects faulty time stamps or missed heartbeats.\(^1\)

• **Heartbeat** is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored. A special case of heartbeat is when the process being monitored periodically resets the watchdog timer in its monitor to prevent it from expiring and thus signaling a fault. For systems where scalability is a concern, transport and processing overhead can be reduced by piggybacking heartbeat messages on to other control messages being exchanged between the process being monitored and the distributed system controller. The big difference between heartbeat and ping/echo is who holds the responsibility for initiating the health check—the monitor or the component itself.

• **Time stamp.** This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A time stamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Simple sequence numbers can also be used for this purpose, if time information is not important.

• **Sanity checking** checks the validity or reasonableness of specific operations or outputs of a component. This tactic is typically based on a knowledge of the internal design, the state of the system, or the nature of the information under scrutiny. It is most often employed at interfaces, to examine a specific information flow.

• **Condition monitoring** involves checking conditions in a process or device, or validating assumptions made during the design. By monitoring conditions, this tactic prevents a system from producing faulty behavior. The computation of checksums is a common example of this tactic. However, the monitor must itself be simple (and, ideally, provable) to
ensure that it does not introduce new software errors.

- **Voting.** The most common realization of this tactic is referred to as triple modular redundancy (TMR), which employs three components that do the same thing, each of which receives identical inputs, and forwards their output to voting logic, used to detect any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault. It must also decide what output to use. It can let the majority rule, or choose some computed average of the disparate outputs. This tactic depends critically on the voting logic, which is usually realized as a simple, rigorously reviewed and tested singleton so that the probability of error is low.

- **Replication** is the simplest form of voting; here, the components are exact clones of each other. Having multiple copies of identical components can be effective in protecting against random failures of hardware, but this cannot protect against design or implementation errors, in hardware or software, because there is no form of diversity embedded in this tactic.

- **Functional redundancy** is a form of voting intended to address the issue of common-mode failures (design or implementation faults) in hardware or software components. Here, the components must always give the same output given the same input, but they are diversely designed and diversely implemented.

- **Analytic redundancy** permits not only diversity among components’ private sides, but also diversity among the components’ inputs and outputs. This tactic is intended to tolerate specification errors by using separate requirement specifications. In embedded systems, analytic redundancy also helps when some input sources are likely to be unavailable at times. For example, avionics programs have multiple ways to compute aircraft altitude, such as using barometric pressure, the radar altimeter, and geometrically using the straight-line distance and look-down angle of a point ahead on the ground. The voter mechanism used with analytic redundancy needs to be more sophisticated than just letting majority rule or computing a simple average. It may have to understand which sensors are currently reliable or not, and it may be asked to produce a higher-fidelity value than any individual component can, by blending and smoothing individual values over time.
• *Exception detection* refers to the detection of a system condition that alters the normal flow of execution. The exception detection tactic can be further refined:

  • *System exceptions* will vary according to the processor hardware architecture employed and include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth.

  • The *parameter fence* tactic incorporates an *a priori* data pattern (such as 0xDEADBEEF) placed immediately after any variable-length parameters of an object. This allows for runtime detection of overwriting the memory allocated for the object’s variable-length parameters.

  • *Parameter typing* employs a base class that defines functions that add, find, and iterate over type-length-value (TLV) formatted message parameters. Derived classes use the base class functions to implement functions that provide parameter typing according to each parameter’s structure. Use of strong typing to build and parse messages results in higher availability than implementations that simply treat messages as byte buckets. Of course, all design involves tradeoffs. When you employ strong typing, you typically trade higher availability against ease of evolution.

  • *Timeout* is a tactic that raises an exception when a component detects that it or another component has failed to meet its timing constraints. For example, a component awaiting a response from another component can raise an exception if the wait time exceeds a certain value.

• *Self-test.* Components (or, more likely, whole subsystems) can run procedures to test themselves for correct operation. Self-test procedures can be initiated by the component itself, or invoked from time to time by a system monitor. These may involve employing some of the techniques found in condition monitoring, such as checksums.

**Recover from Faults**

Recover-from-faults tactics are refined into *preparation-and-repair* tactics and *reintroduction* tactics. The latter are concerned with reintroducing a failed (but rehabilitated) component back into normal operation.

Preparation-and-repair tactics are based on a variety of combinations of retrying a computation or introducing redundancy. They include the following:
• **Active redundancy (hot spare).** This refers to a configuration where all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Because the redundant spare possesses an identical state to the active processor, it can take over from a failed component in a matter of milliseconds. The simple case of one active node and one redundant spare node is commonly referred to as 1+1 (“one plus one”) redundancy. Active redundancy can also be used for facilities protection, where active and standby network links are used to ensure highly available network connectivity.

• **Passive redundancy (warm spare).** This refers to a configuration where only the active members of the protection group process input traffic; one of their duties is to provide the redundant spare(s) with periodic state updates. Because the state maintained by the redundant spares is only loosely coupled with that of the active node(s) in the protection group (with the looseness of the coupling being a function of the checkpointing mechanism employed between active and redundant nodes), the redundant nodes are referred to as warm spares. Depending on a system’s availability requirements, passive redundancy provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy tactic and the less available but significantly less complex cold spare tactic (which is also significantly cheaper). (For an example of implementing passive redundancy, see the section on code templates in *Chapter 19.*

• **Spare (cold spare).** Cold sparing refers to a configuration where the redundant spares of a protection group remain out of service until a failover occurs, at which point a power-on-reset procedure is initiated on the redundant spare prior to its being placed in service. Due to its poor recovery performance, cold sparing is better suited for systems having only high-reliability (MTBF) requirements as opposed to those also having high-availability requirements.

• **Exception handling.** Once an exception has been detected, the system must handle it in some fashion. The easiest thing it can do is simply to crash, but of course that’s a terrible idea from the point of availability, usability, testability, and plain good sense. There are much more productive possibilities. The mechanism employed for exception handling depends largely on the programming environment employed, ranging from simple function return codes (error codes) to the use of exception
classes that contain information helpful in fault correlation, such as the
name of the exception thrown, the origin of the exception, and the cause
of the exception thrown. Software can then use this information to mask
the fault, usually by correcting the cause of the exception and retrying the
operation.

• Rollback. This tactic permits the system to revert to a previous known
good state, referred to as the “rollback line”—rolling back time—upon the
detection of a failure. Once the good state is reached, then execution can
continue. This tactic is often combined with active or passive redundancy
tactics so that after a rollback has occurred, a standby version of the failed
component is promoted to active status. Rollback depends on a copy of a
previous good state (a checkpoint) being available to the components that
are rolling back. Checkpoints can be stored in a fixed location and
updated at regular intervals, or at convenient or significant times in the
processing, such as at the completion of a complex operation.

• Software upgrade is another preparation-and-repair tactic whose goal is to
achieve in-service upgrades to executable code images in a non-service-
affecting manner. This may be realized as a function patch, a class patch,
or a hitless in-service software upgrade (ISSU). A function patch is used
in procedural programming and employs an incremental linker/loader to
store an updated software function into a pre-allocated segment of target
memory. The new version of the software function will employ the entry
and exit points of the deprecated function. Also, upon loading the new
software function, the symbol table must be updated and the instruction
cache invalidated. The class patch tactic is applicable for targets executing
object-oriented code, where the class definitions include a back-door
mechanism that enables the runtime addition of member data and
functions. Hitless in-service software upgrade leverages the active
redundancy or passive redundancy tactics to achieve non-service-affecting
upgrades to software and associated schema. In practice, the function
patch and class patch are used to deliver bug fixes, while the hitless in-
service software upgrade is used to deliver new features and capabilities.

• Retry. The retry tactic assumes that the fault that caused a failure is
transient and retrying the operation may lead to success. This tactic is
used in networks and in server farms where failures are expected and
common. There should be a limit on the number of retries that are
attempted before a permanent failure is declared.

• Ignore faulty behavior. This tactic calls for ignoring messages sent from a
particular source when we determine that those messages are spurious. For example, we would like to ignore the messages of an external component launching a denial-of-service attack by establishing Access Control List filters, for example.

• The *degradation* tactic maintains the most critical system functions in the presence of component failures, dropping less critical functions. This is done in circumstances where individual component failures gracefully reduce system functionality rather than causing a complete system failure.

• *Reconfiguration* attempts to recover from component failures by reassigning responsibilities to the (potentially restricted) resources left functioning, while maintaining as much functionality as possible.

Reintroduction is where a failed component is reintroduced after it has been corrected. Reintroduction tactics include the following:

• The *shadow* tactic refers to operating a previously failed or in-service upgraded component in a “shadow mode” for a predefined duration of time prior to reverting the component back to an active role. During this duration its behavior can be monitored for correctness and it can repopulate its state incrementally.

• *State resynchronization* is a reintroduction partner to the active redundancy and passive redundancy preparation-and-repair tactics. When used alongside the active redundancy tactic, the state resynchronization occurs organically, because the active and standby components each receive and process identical inputs in parallel. In practice, the states of the active and standby components are periodically compared to ensure synchronization. This comparison may be based on a cyclic redundancy check calculation (checksum) or, for systems providing safety-critical services, a message digest calculation (a one-way hash function). When used alongside the passive redundancy (warm spare) tactic, state resynchronization is based solely on periodic state information transmitted from the active component(s) to the standby component(s), typically via checkpointing. A special case of this tactic is found in stateless services, whereby any resource can handle a request from another (failed) resource.

• *Escalating restart* is a reintroduction tactic that allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected. For example, consider a system that supports four levels of restart, as follows. The lowest level of restart (call it Level 0), and hence having the least impact
on services, employs passive redundancy (warm spare), where all child threads of the faulty component are killed and recreated. In this way, only data associated with the child threads is freed and reinitialized. The next level of restart (Level 1) frees and reinitializes all unprotected memory (protected memory would remain untouched). The next level of restart (Level 2) frees and reinitializes all memory, both protected and unprotected, forcing all applications to reload and reinitialize. And the final level of restart (Level 3) would involve completely reloading and reinitializing the executable image and associated data segments. Support for the escalating restart tactic is particularly useful for the concept of graceful degradation, where a system is able to degrade the services it provides while maintaining support for mission-critical or safety-critical applications.

• Non-stop forwarding (NSF) is a concept that originated in router design. In this design functionality is split into two parts: supervisory, or control plane (which manages connectivity and routing information), and data plane (which does the actual work of routing packets from sender to receiver). If a router experiences the failure of an active supervisor, it can continue forwarding packets along known routes—with neighboring routers—while the routing protocol information is recovered and validated. When the control plane is restarted, it implements what is sometimes called “graceful restart,” incrementally rebuilding its routing protocol database even as the data plane continues to operate.

Prevent Faults
Instead of detecting faults and then trying to recover from them, what if your system could prevent them from occurring in the first place? Although this sounds like some measure of clairvoyance might be required, it turns out that in many cases it is possible to do just that. ³

• Removal from service. This tactic refers to temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures. One example involves taking a component of a system out of service and resetting the component in order to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults affects service (resulting in system failure). Another term for this tactic is software rejuvenation.

• Transactions. Systems targeting high-availability services leverage
transactional semantics to ensure that asynchronous messages exchanged between distributed components are *atomic, consistent, isolated, and durable*. These four properties are called the “ACID properties.” The most common realization of the transactions tactic is “two-phase commit” (a.k.a. 2PC) protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item.

- **Predictive model.** A predictive model, when combined with a monitor, is employed to monitor the state of health of a system process to ensure that the system is operating within its nominal operating parameters, and to take corrective action when conditions are detected that are predictive of likely future faults. The operational performance metrics monitored are used to predict the onset of faults; examples include session establishment rate (in an HTTP server), threshold crossing (monitoring high and low water marks for some constrained, shared resource), or maintaining statistics for process state (in service, out of service, under maintenance, idle), message queue length statistics, and so on.

- **Exception prevention.** This tactic refers to techniques employed for the purpose of preventing system exceptions from occurring. The use of exception classes, which allows a system to transparently recover from system exceptions, was discussed previously. Other examples of exception prevention include abstract data types, such as smart pointers, and the use of wrappers to prevent faults, such as dangling pointers and semaphore access violations from occurring. Smart pointers prevent exceptions by doing bounds checking on pointers, and by ensuring that resources are automatically deallocated when no data refers to it. In this way resource leaks are avoided.

- **Increase competence set.** A program’s competence set is the set of states in which it is “competent” to operate. For example, the state when the denominator is zero is outside the competence set of most divide programs. When a component raises an exception, it is signaling that it has discovered itself to be outside its competence set; in essence, it doesn’t know what to do and is throwing in the towel. Increasing a component’s competence set means designing it to handle more cases—faults—as part of its normal operation. For example, a component that assumes it has access to a shared resource might throw an exception if it discovers that access is blocked. Another component might simply wait for access, or return immediately with an indication that it will complete its operation on its own the next time it does have access. In this example,
the second component has a larger competence set than the first.

5.3. A Design Checklist for Availability

Table 5.4 is a checklist to support the design and analysis process for availability.

Table 5.4. Checklist to Support the Design and Analysis Process for Availability

<table>
<thead>
<tr>
<th>Category</th>
<th>Checklist</th>
</tr>
</thead>
</table>
| Allocation of Responsibilities| Determine the system responsibilities that need to be highly available. Within those responsibilities, ensure that additional responsibilities have been allocated to detect an omission, crash, incorrect timing, or incorrect response. Additionally, ensure that there are responsibilities to do the following:  
  ▪ Log the fault  
  ▪ Notify appropriate entities (people or systems)  
  ▪ Disable the source of events causing the fault  
  ▪ Be temporarily unavailable  
  ▪ Fix or mask the fault/failure  
  ▪ Operate in a degraded mode |
| Coordination Model           | Determine the system responsibilities that need to be highly available. With respect to those responsibilities, do the following:  
  ▪ Ensure that coordination mechanisms can detect an omission, crash, incorrect timing, or incorrect response. Consider, for example, whether guaranteed delivery is necessary. Will the coordination work under conditions of degraded communication?  
  ▪ Ensure that coordination mechanisms enable the logging of the fault, notification of appropriate entities, disabling of the source of the events causing the fault, fixing or masking the fault, or operating in a degraded mode.  
  ▪ Ensure that the coordination model supports the replacement of the artifacts used (processors, communications channels, persistent storage, and processes). For example, does replacement of a server allow the system to continue to operate? |
- Determine if the coordination will work under conditions of degraded communication, at startup/shutdown, in repair mode, or under overloaded operation. For example, how much lost information can the coordination model withstand and with what consequences?

**Data Model**

Determine which portions of the system need to be highly available. Within those portions, determine which data abstractions, along with their operations or their properties, could cause a fault of omission, a crash, incorrect timing behavior, or an incorrect response.

For those data abstractions, operations, and properties, ensure that they can be disabled, be temporarily unavailable, or be fixed or masked in the event of a fault.

For example, ensure that write requests are cached if a server is temporarily unavailable and performed when the server is returned to service.

**Mapping among Architectural Elements**

Determine which artifacts (processors, communication channels, persistent storage, or processes) may produce a fault: omission, crash, incorrect timing, or incorrect response.

Ensure that the mapping (or remapping) of architectural elements is flexible enough to permit the recovery from the fault. This may involve a consideration of the following:

- Which processes on failed processors need to be reassigned at runtime
- Which processors, data stores, or communication channels can be activated or reassigned at runtime
- How data on failed processors or storage can be served by replacement units
Resource Management

Determine what critical resources are necessary to continue operating in the presence of a fault: omission, crash, incorrect timing, or incorrect response. Ensure there are sufficient remaining resources in the event of a fault to log the fault; notify appropriate entities (people or systems); disable the source of events causing the fault; be temporarily unavailable; fix or mask the fault/failure; operate normally, in startup, shutdown, repair mode, degraded operation, and overloaded operation.

Determine the availability time for critical resources, what critical resources must be available during specified time intervals, time intervals during which the critical resources may be in a degraded mode, and repair time for critical resources. Ensure that the critical resources are available during these time intervals.

For example, ensure that input queues are large enough to buffer anticipated messages if a server fails so that the messages are not permanently lost.
5.4. Summary

Availability refers to the ability of the system to be available for use, especially after a fault occurs. The fault must be recognized (or prevented) and then the system must respond in some fashion. The response desired will depend on the criticality of the application and the type of fault and can range from “ignore it” to “keep on going as if it didn’t occur.”

Tactics for availability are categorized into detect faults, recover from faults, and prevent faults. Detection tactics depend, essentially, on detecting signs of life from various components. Recovery tactics are some combination of retrying an operation or maintaining redundant data or computations. Prevention tactics depend either on removing elements from service or utilizing mechanisms to limit the scope of faults.

All of the availability tactics involve the coordination model because the coordination model must be aware of faults that occur to generate an appropriate response.
5.5. For Further Reading

Patterns for availability:
• You can find patterns for fault tolerance in [Hanmer 07].

Tactics for availability, overall:
• A more detailed discussion of some of the availability tactics in this chapter is given in [Scott 09]. This is the source of much of the material in this chapter.
• The Internet Engineering Task Force has promulgated a number of standards supporting availability tactics. These standards include non-stop forwarding [IETF 04], ping/echo ICMPv6 [IETF 06b], echo request/response), and MPLS (LSP Ping) networks [IETF 06a].

Tactics for availability, fault detection:
• The parameter fence tactic was first used (to our knowledge) in the Control Data Series computers of the late 1960s.
• Triple modular redundancy (TMR), part of the voting tactic, was developed in the early 1960s by Lyons [Lyons 62].
• The fault detection tactic of voting is based on the fundamental contributions to automata theory by Von Neumann, who demonstrated how systems having a prescribed reliability could be built from unreliable components [Von Neumann 56].

Tactics for availability, fault recovery:
• Standards-based realizations of active redundancy exist for protecting network links (i.e., facilities) at both the physical layer [Bellcore 99, Telcordia 00] and the network/link layer [IETF 05].
• Exception handling has been written about by [Powel Douglass 99]. Software can then use this information to mask the fault, usually by correcting the cause of the exception and retrying the operation.
• [Morelos-Zaragoza 06] and [Schneier 96] have written about the comparison of state during resynchronization.
• Some examples of how a system can degrade through use (degradation) are given in [Nygard 07].
• [Utas 05] has written about escalating restart.
• Mountains of papers have been written about parameter typing, but [Utas 05] writes about it in the context of availability (as opposed to bug prevention, its usual context).
Hardware engineers often use preparation-and-repair tactics. Examples include error detection and correction (EDAC) coding, forward error correction (FEC), and temporal redundancy. EDAC coding is typically used to protect control memory structures in high-availability distributed real-time embedded systems [Hamming 80]. Conversely, FEC coding is typically employed to recover from physical-layer errors occurring on external network links Morelos-Zaragoza 06]. Temporal redundancy involves sampling spatially redundant clock or data lines at time intervals that exceed the pulse width of any transient pulse to be tolerated, and then voting out any defects detected [Mavis 02].

Tactics for availability, fault prevention:

- Parnas and Madey have written about increasing an element’s competence set [Parnas 95].
- The ACID properties, important in the transactions tactic, were introduced by Gray in the 1970s and discussed in depth in [Gray 93].

Analysis:

- Fault tree analysis dates from the early 1960s, but the granddaddy of resources for it is the U.S. Nuclear Regulatory Commission’s “Fault Tree Handbook,” published in 1981 [Vesely 81]. NASA’s 2002 “Fault Tree Handbook with Aerospace Applications” [Vesely 02] is an updated comprehensive primer of the NRC handbook, and the source for the notation used in this chapter. Both are available online as downloadable PDF files.

5.6. Discussion Questions

1. Write a set of concrete scenarios for availability using each of the possible responses in the general scenario.
2. Write a concrete availability scenario for the software for a (hypothetical) pilotless passenger aircraft.
3. Write a concrete availability scenario for a program like Microsoft Word.
4. Redundancy is often cited as a key strategy for achieving high availability. Look at the tactics presented in this chapter and decide how many of them exploit some form of redundancy and how many do not.
5. How does availability trade off against modifiability? How would you make a change to a system that is required to have “24/7” availability (no scheduled or unscheduled downtime, ever)?
6. Create a fault tree for an automatic teller machine. Include faults dealing with hardware component failure, communications failure, software failure, running out of supplies, user errors, and security attacks. How would you modify your automatic teller machine design to accommodate these faults?

7. Consider the fault detection tactics (ping/echo, heartbeat, system monitor, voting, and exception detection). What are the performance implications of using these tactics?