4

Availability

Technology does not always rhyme
with perfection and reliability.
Far from it in reality!

—Jean-Michel Jarre

Availability refers to a property of software—namely, 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). Availability builds on 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, as we'll see in this chapter.

Availability also encompasses the ability of a system to mask or repair faults such that they do not become failures, thereby ensuring that the cumulative service outage period does not exceed a required value over a specified time interval. This definition subsumes concepts of reliability, robustness, and any other quality attribute that involves a concept of unacceptable failure.

A failure is the deviation of the system from its specification, where that deviation is externally visible. Determining that a failure has occurred requires some external observer in the environment.

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. Through these

actions, 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.

Availability is closely related to, but clearly distinct from, 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, since it may be difficult to tell when a system has failed and when it is simply being egregiously 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.

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. Once those are understood, mitigation strategies can be designed into the system.

Since a system failure is observable by users, the time to repair is the time until the failure is no longer observable. This may be an imperceptible delay in a user’s 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” is critical here: If a failure *could have been observed*, then it is a failure, whether or not it was actually observed.

In addition, we are often concerned with the level of capability that remains when a failure has occurred—a degraded operating mode.

Distinguishing between faults and failures allows us to discuss repair strategies. If code containing a fault is executed but the system is able to recover from the fault without any observable deviation from the otherwise specified behavior, we say that no failure has occurred.

The availability of a system can be measured as the probability that it will provide the specified services within the required bounds over a specified time interval. A well-known expression is used to derive steady-state availability (which came from the world of hardware):

$$MTBF/(MTBF + MTTR)$$

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 it is that such an event will occur, and how much time will be required to repair it.

From this formula, it is possible to calculate probabilities and make claims like "the system exhibits 99.999 percent availability" or "there is 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) should not be considered when calculating availability, since the system is deemed "not needed" then; of course, this is dependent on the specific requirements for the system, which are often encoded in a service level agreement (SLA). This 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.

Detected faults can be categorized prior to being reported and repaired. This categorization is commonly based on the fault's severity (critical, major, or minor) and service impact (service-affecting or non-service-affecting). It provides the system operator with a timely and accurate system status and allows for an appropriate repair strategy to be employed. The repair strategy may be automated or may require manual intervention.

As just mentioned, the availability expected of a system or service is frequently expressed as an SLA. The SLA specifies the availability level that is guaranteed and, usually, the penalties that the provider will suffer if the SLA is violated. For example, Amazon provides the following SLA for its EC2 cloud service:

AWS will use commercially reasonable efforts to make the Included Services each available for each AWS region with a Monthly Uptime Percentage of at least 99.99%, in each case during any monthly billing cycle (the “Service Commitment”). In the event any of the Included Services do not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

Table 4.1 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. As mentioned earlier, only unscheduled outages contribute to system downtime.

Table 4.1 System Availability Requirements

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hr, 36 min	3 days, 15.6 hr
99.9%	2 hr, 10 min	8 hr, 45 min.
99.99%	12 min, 58 sec	52 min, 34 sec
99.999%	1 min, 18 sec	5 min, 15 sec
99.9999%	8 sec	32 sec

4.1 Availability General Scenario

We can now describe the individual portions of an availability general scenario as summarized in Table 4.2.

Table 4.2 Availability General Scenario

Portion of Scenario	Description	Possible Values
Source	This specifies where the fault comes from.	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	The stimulus to an availability scenario is a fault.	Fault: omission, crash, incorrect timing, incorrect response
Artifact	This specifies which portions of the system are responsible for and affected by the fault.	Processors, communication channels, storage, processes, affected artifacts in the system's environment
Environment	We may be interested in not only how a system behaves in its "normal" environment, but also how it behaves in situations such as when it is already recovering from a fault.	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation

Portion of Scenario	Description	Possible Values
Response	The most commonly desired response is to prevent the fault from becoming a failure, but other responses may also be important, such as notifying people or logging the fault for later analysis. This section specifies the desired system response.	Prevent the fault from becoming a failure Detect the fault: • Log the fault • Notify the appropriate entities (people or systems) • Recover from the fault • Disable the source of events causing the fault • Be temporarily unavailable while a repair is being effected • Fix or mask the fault/failure or contain the damage it causes • Operate in a degraded mode while a repair is being effected • Time or time interval when the system must be available • Availability percentage (e.g., 99.999 percent) • Time to detect the fault

Portion of Scenario	Description	Possible Values
Response measure	We may focus on a number of measures of availability, depending on the criticality of the service being provided.	• Time to repair the fault • Time or time interval in which system can be in degraded mode • Proportion (e.g., 99 percent) or rate (e.g., up to 100 per second) of a certain class of faults that the system prevents, or handles without failing

An example concrete availability scenario derived from the general scenario in Table 4.2 is shown in Figure 4.1. The scenario is this: *A server in a server farm fails during normal operation, and the system informs the operator and continues to operate with no downtime.*

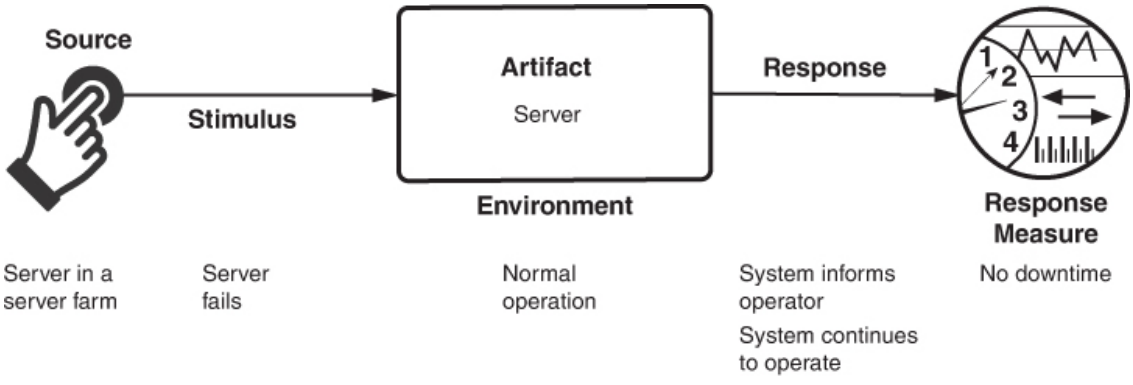

Figure 4.1 Sample concrete availability scenario

4.2 Tactics for Availability

A failure occurs when the system no longer delivers a service that is consistent with its specification and this failure is observable by the system's actors. A fault (or combination of faults) has the potential to cause a failure. Availability tactics, in turn, are designed to enable a system to prevent or 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, as illustrated in Figure 4.2.

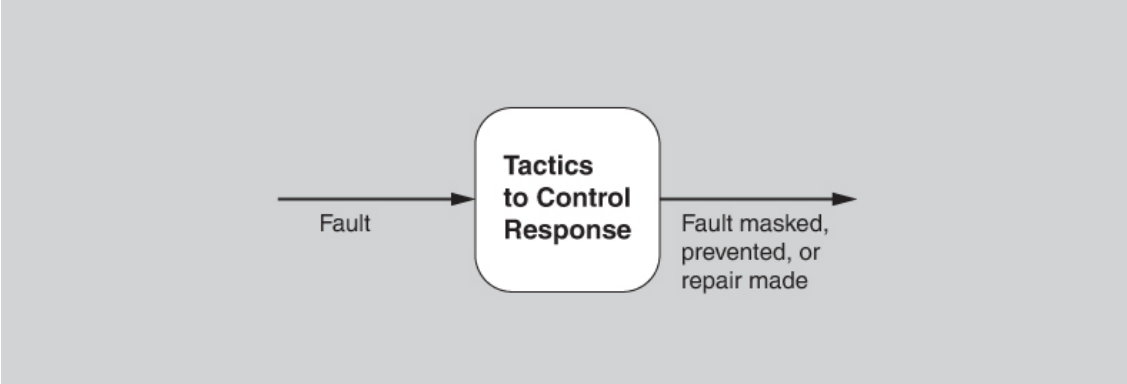

Figure 4.2 Goal of availability tactics

Availability tactics have one of three purposes: fault detection, fault recovery, or fault prevention. The tactics for availability are shown in Figure 4.3. These tactics will often be provided by a software infrastructure, such as a middleware package, so your job as an architect may be choosing and assessing (rather than implementing) the right availability tactics and the right combination of tactics.

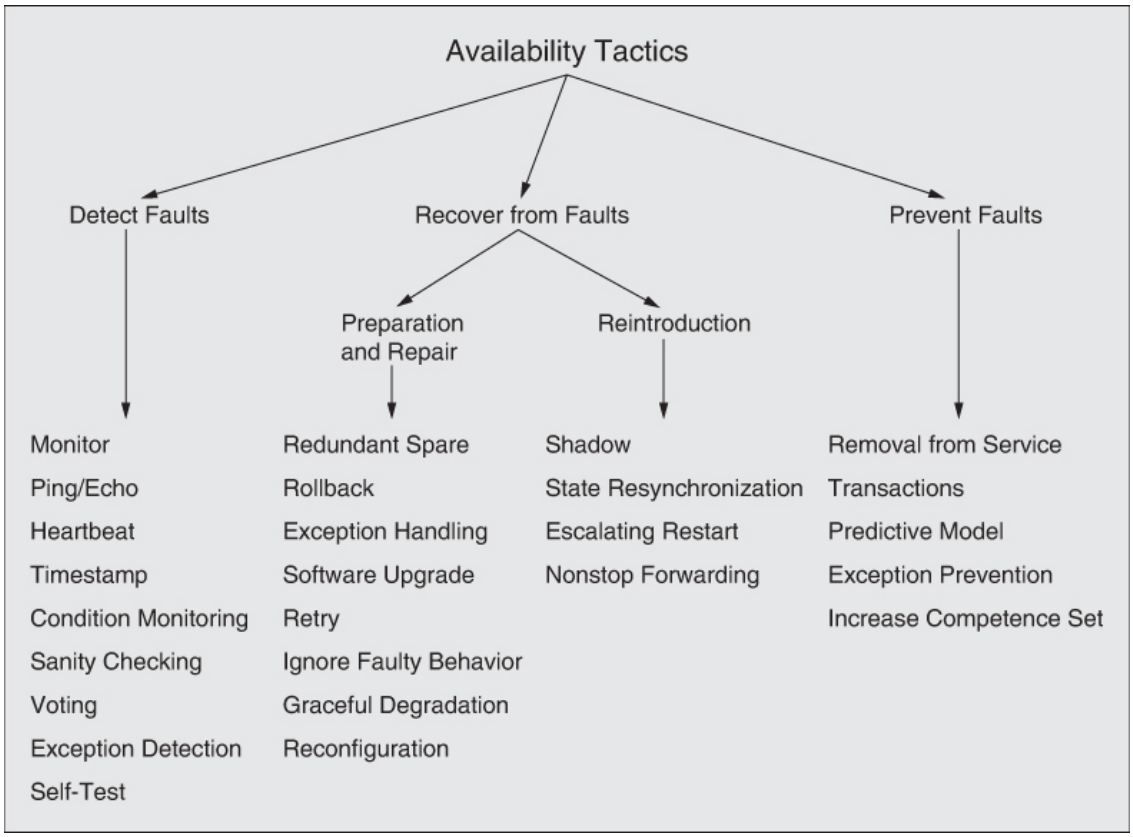

Figure 4.3 Availability tactics

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:

- *Monitor*. This component is used to monitor the state of health of various other parts of the system: processors, processes, I/O, memory, and so forth. 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 *timestamps* or missed *heartbeats*.[1]

---

1. When the detection mechanism is implemented using a counter or timer that is periodically reset, this specialization of the system monitor is referred to as a *watchdog*. During nominal operation, the process being monitored will periodically reset the watchdog counter/timer as part of its signal that it's working correctly; this is sometimes referred to as "petting the watchdog."

- *Ping/echo.* In this tactic, an asynchronous request/response message pair is exchanged between nodes; it is used to determine reachability and the round-trip delay through the associated network path. In addition, the echo indicates that the pinged component is alive. 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 Internet Protocol (IP).
- *Heartbeat.* This fault detection mechanism 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 onto other control messages being exchanged. The difference between heartbeat and ping/echo lies in who holds the responsibility for initiating the health check—the monitor or the component itself.
- *Timestamp.* This tactic is used to detect incorrect sequences of events, primarily in distributed message-passing systems. A timestamp of an event can be established by assigning the state of a local clock to the event immediately after the event occurs. Sequence numbers can also be used for this purpose, since timestamps in a distributed system may be inconsistent across different processors. See Chapter 17 for a fuller discussion of the topic of time in a distributed system.

- *Condition monitoring*. This tactic 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, provably correct) to ensure that it does not introduce new software errors.
- *Sanity checking*. This tactic checks the validity or reasonableness of specific operations or outputs of a component. It 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.
- *Voting*. Voting involves comparing computational results from multiple sources that should be producing the same results and, if they are not, deciding which results to use. 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. Voting also depends critically on having multiple sources to evaluate. Typical schemes include the following:
  - *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 cannot protect against design or implementation errors, in hardware or software, since there is no form of diversity embedded in this tactic.
  - *Functional redundancy*, in contrast, is intended to address the issue of common-mode failures (where replicas exhibit the same fault at the same time because they share the same implementation) in hardware or software components, by implementing design diversity. This tactic attempts to deal with the systematic nature of design faults by adding diversity to redundancy. The outputs of functionally redundant components should be the same given the same input. The functional redundancy tactic is still vulnerable to specification errors—and, of course, functional replicas will be more expensive to develop and verify.

- *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 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, with 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.* This tactic focuses on the detection of a system condition that alters the normal flow of execution. It can be further refined as follows:
  - *System exceptions* will vary according to the processor hardware architecture employed. They include faults such as divide by zero, bus and address faults, illegal program instructions, and so forth.
  - The *parameter fence* tactic incorporates a known 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 provide functions to build and parse messages. Use of parameter typing ensures that the sender and the receiver of messages agree on the type of the content, and detects cases where they don't.
  - *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 an-

other 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:

- *Redundant spare.* This tactic refers to a configuration in which one or more duplicate components can step in and take over the work if the primary component fails. This tactic is at the heart of the hot spare, warm spare, and cold spare patterns, which differ primarily in how up-to-date the backup component is at the time of its takeover.
- *Rollback.* A rollback 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 the transactions tactic and the redundant spare tactic 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.
- *Exception handling.* Once an exception has been detected, the system will 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, the origin of the exception, and the cause of the exception Software can then use this information to mask or repair the fault.

- *Software upgrade.* The goal of this tactic is to achieve in-service upgrades to executable code images in a non-service-affecting manner. Strategies include the following:
  - *Function patch*. This kind of patch, which is used in procedural programming, 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.
  - *Class patch*. This kind of upgrade is applicable for targets executing object-oriented code, where the class definitions include a backdoor mechanism that enables the runtime addition of member data and functions.
  - *Hitless in-service software upgrade (ISSU)*. This leverages the redundant spare tactic 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 ISSU is used to deliver new features and capabilities.
- *Retry.* The retry tactic assumes that the fault that caused a failure is transient, and that retrying the operation may lead to success. It is used in networks and in server farms where failures are expected and common. A limit should be placed 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 emanating from the live failure of a sensor.
- *Graceful degradation.* This tactic maintains the most critical system functions in the presence of component failures, while 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.* Reconfiguration attempts to recover from failures by reassigning responsibilities to the (potentially restricted) resources or components left functioning, while maintaining as much functionality as possible.

Reintroduction occurs when a failed component is reintroduced after it has been repaired. Reintroduction tactics include the following:

- *Shadow.* This 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.* This reintroduction tactic is a partner to the redundant spare tactic. When used with active redundancy—a version of the redundant spare tactic—the state resynchronization occurs organically, since 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 version of the redundant 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.
- *Escalating restart.* This reintroduction tactic allows the system to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affectation. For example, consider a system that supports four levels of restart, numbered 0–3. The lowest level of restart (Level 0) has the least impact on services and 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 is untouched. The next level of restart (Level 2) frees and reinitializes all memory, both protected and unprotected, forcing all applications to reload and reinitialize. The final level of restart (Level 3) involves 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.

- *Nonstop forwarding.* This concept originated in router design, and assumes that functionality is split into two parts: the supervisory or control plane (which manages connectivity and routing information) and the 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 a "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 it might sound as if some measure of clairvoyance would be required, it turns out that in many cases it is possible to do just that.[2]

---

2. These tactics deal with runtime means to prevent faults from occurring. Of course, an excellent way to prevent faults—at least in the system you're building, if not in systems that your system must interact with—is to produce high-quality code. This can be done by means of code inspections, pair programming, solid requirements reviews, and a host of other good engineering practices.

- *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. For example, a component of a system might be taken out of service and reset to scrub latent faults (such as memory leaks, fragmentation, or soft errors in an unprotected cache) before the accumulation of faults reaches the service-affecting level, resulting in system failure. Other terms for this tactic are *software rejuvenation* and *therapeutic reboot.* If you reboot your computer every night, you are practicing removal from service.
- *Transactions*. Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are atomic, consistent, isolated, and durable—properties collectively referred to as the "ACID properties." The most common realization of the transactions tactic is the "two-phase commit" (2PC) protocol. This tactic prevents race conditions caused by two processes attempting to update the same data item at the same time.
- *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 the system nears a critical threshold. The operational performance metrics monitored are used to predict the onset of faults; examples include the session establishment rate (in an HTTP server), threshold crossing (monitoring high and low watermarks for some constrained, shared resource), statistics on the process state (e.g., in-service, out-of-service, under maintenance, idle), and message queue length statistics.
- *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 earlier. Other examples of exception prevention include error-correcting code (used in telecommunications), abstract data types such as smart pointers, and the use of wrappers to prevent faults such as dangling pointers or semaphore access violations. Smart pointers prevent exceptions by doing bounds checking on pointers, and by ensuring that resources are automati-

cally de-allocated when no data refers to them, thereby avoiding resource leaks.

- *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.

4.3 Tactics-Based Questionnaire for Availability

Based on the tactics described in Section 4.2, we can create a set of availability tactics–inspired questions, as presented in Table 4.3. To gain an overview of the architectural choices made to support availability, the analyst asks each question and records the answers in the table. The answers to these questions can then be made the focus of further activities: investigation of documentation, analysis of code or other artifacts, reverse engineering of code, and so forth.

Table 4.3 Tactics-Based Questionnaire for Availability

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Detect Faults	Does the system use				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	ping/echo to detect failure of a component or connection, or network congestion?				
	Does the system use a component to **monitor** the state of health of other parts of the system? A system monitor can detect failure or congestion in the network or other shared resources, such as from a denial-of-service attack.				
	Does the system use a				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	heartbeat—a periodic message exchange between a system monitor and a process—to detect failure of a component or connection, or network congestion?				
	Does the system use a **timestamp** to detect incorrect sequences of events in distributed systems?				
	Does the system use **voting** to check that replicated components are				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	producing the same results?				
	The replicated components may be identical replicas, functionally redundant, or analytically redundant.				
	Does the system use **exception detection** to detect a system condition that alters the normal flow of execution (e.g., system exception, parameter fence, parameter typing, timeout)?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Can the system do a **self-test** to test itself for correct operation?				
Recover from Faults (Preparation and Repair)	Does the system employ **redundant spares**? Is a component's role as active versus spare fixed, or does it change in the presence of a fault? What is the switchover mechanism? What is the trigger for a switchover? How long does it take for a spare to assume its duties?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system employ **exception handling** to deal with faults? Typically the handling involves either reporting, correcting, or masking the fault.				
	Does the system employ **rollback**, so that it can revert to a previously saved good state (the “rollback line”) in the event of a fault?				
	Can the system perform in-service **software upgrades** to ex-				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	ecutable code images in a non-service-affecting manner?				
	Does the system systematically **retry** in cases where the component or connection failure may be transient?				
	Can the system simply ignore faulty behavior (e.g., ignore messages when it is determined that those messages are spurious)?				
	Does the system have a				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	policy of degradation when resources are compromised, maintaining the most critical system functions in the presence of component failures, and dropping less critical functions?				
	Does the system have consistent policies and mechanisms for **reconfiguration** after failures, reassigning responsibilities to the resources left functioning, while main-				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	taining as much functionality as possible?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Recover from Faults (Reintroduction)	Can the system operate a previously failed or in-service upgraded component in a "**shadow** mode" for a predefined time prior to reverting the component back to an active role?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	If the system uses active or passive redundancy, does it also employ **state resynchronization** to send state information from active components to standby components?				
	Does the system employ **escalating restart** to recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Can message processing and routing portions of the system employ **non-stop forwarding**, where functionality is split into supervisory and data planes?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
Prevent Faults	Can the system **remove components from service**, temporarily placing a system component in an out-of-service state for the purpose of preempting potential system failures?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	Does the system employ **transactions**—bundling state updates so that asynchronous messages exchanged between distributed components are *atomic*, *consistent*, *isolated*, and *durable*?				
	Does the system use a **predictive model** to monitor the state of health of a component to ensure that the system is operating within nominal parameters?				

Tactics Group	Tactics Question	Support? (Y/N)	Risk	Design Decisions and Location	Rationale and Assumptions
	When conditions are detected that are predictive of likely future faults, the model initiates corrective action.				

4.4 Patterns for Availability

This section presents a few of the most important architectural patterns for availability.

The first three patterns are all centered on the redundant spare tactic, and will be described as a group. They differ primarily in the degree to which the backup components' state matches that of the active component. (A special case occurs when the components are stateless, in which case the first two patterns become identical.)

- *Active redundancy (hot spare)*. For stateful components, this refers to a configuration in which all of the nodes (active or redundant spare) in a protection group[3] receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain a 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 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.

---

3. A protection group is a group of processing nodes in which one or more nodes are “active,” with the remaining nodes serving as redundant spares.

- *Passive redundancy (warm spare)*. For stateful components, this refers to a configuration in which 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 period of the state updates), the redundant nodes are referred to as warm spares. Passive redundancy provides a solution that achieves a balance between the more highly available but more compute-intensive (and expensive) active redundancy pattern and the less available but significantly less complex cold spare pattern (which is also significantly cheaper).
- *Spare (cold spare)*. Cold sparing refers to a configuration in which redundant spares remain out of service until a failover occurs, at which point a power-on-reset[4] procedure is initiated on the redundant spare prior to its being placed in service. Due to its poor recovery performance, and hence its high mean time to repair, this pattern is poorly suited to systems having high-availability requirements.

---

4. A power-on-reset ensures that a device starts operating in a known state.

Benefits:

- The benefit of a redundant spare is a system that continues to function correctly after only a brief delay in the presence of a failure. The alternative is a system that stops functioning correctly, or stops functioning altogether, until the failed component is repaired. This repair could take hours or days.

Tradeoffs:

- The tradeoff with any of these patterns is the additional cost and complexity incurred in providing a spare.
- The tradeoff among the three alternatives is the time to recover from a failure versus the runtime cost incurred to keep a spare up-to-date. A hot spare carries the highest cost but leads to the fastest recovery time, for example.

Other patterns for availability include the following.

- *Triple modular redundancy (TMR)*. This widely used implementation of the voting tactic employs three components that do the same thing. Each component receives identical inputs and forwards its output to the voting logic, which detects any inconsistency among the three output states. Faced with an inconsistency, the voter reports a fault. It must also decide which output to use, and different instantiations of this pattern use different decision rules. Typical choices are letting the majority rule or choosing some computed average of the disparate outputs.

  Of course, other versions of this pattern that employ 5 or 19 or 53 redundant components are also possible. However, in most cases, 3 components are sufficient to ensure a reliable result.

  Benefits:

  - TMR is simple to understand and to implement. It is blissfully independent of what might be causing disparate results, and is only concerned about making a reasonable choice so that the system can continue to function.

  Tradeoffs:

  - There is a tradeoff between increasing the level of replication, which raises the cost, and the resulting availability. In systems employing TMR, the statistical likelihood of two or more components failing is vanishingly small, and three components represents a sweet spot between availability and cost.
- *Circuit breaker*. A commonly used availability tactic is retry. In the event of a timeout or fault when invoking a service, the invoker simply tries again—and again, and again. A circuit breaker keeps the invoker from trying countless times, waiting for a response that never comes. In this way, it breaks the endless retry cycle when it deems that

the system is dealing with a fault. That's the signal for the system to begin handling the fault. Until the circuit break is "reset," subsequent invocations will return immediately without passing along the service request.

Benefits:

- This pattern can remove from individual components the policy about how many retries to allow before declaring a failure.
- At worst, endless fruitless retries would make the invoking component as useless as the invoked component that has failed. This problem is especially acute in distributed systems, where you could have many callers calling an unresponsive component and effectively going out of service themselves, causing the failure to cascade across the whole system. The circuit breaker, *in conjunction with software that listens to it and begins recovery procedures*, prevents that problem.

Tradeoffs:

- Care must be taken in choosing timeout (or retry) values. If the timeout is too long, then unnecessary latency is added. But if the timeout is too short, then the circuit breaker will be tripping when it does not need to—a kind of "false positive"—which can lower the availability and performance of these services.

Other availability patterns that are commonly used include the following:

- *Process pairs*. This pattern employs checkpointing and rollback. In case of failure, the backup has been checkpointing and (if necessary) rolling back to a safe state, so is ready to take over when a failure occurs.
- *Forward error recovery*. This pattern provides a way to get out of an undesirable state by *moving forward* to a desirable state. This often relies upon built-in error-correction capabilities, such as data redundancy, so that errors may be corrected without the need to fall back to a previous state or to retry. Forward error recovery finds a safe, possibly degraded state from which the operation can move forward.

4.5 For Further Reading

Patterns for availability:

- You can read about patterns for fault tolerance in [Hanmer 13].

General tactics for availability:

- 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 2004], *Ping/Echo* (*ICMP* [IETF 1981] or *ICMPv6* [RFC 2006b] *Echo Request/Response*), and MPLS (LSP Ping) networks [IETF 2006a].

Tactics for availability—fault detection:

- Triple modular redundancy (TMR) was developed in the early 1960s by Lyons [Lyons 62].
- The fault detection in the voting tactic 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 of the seven-layer OSI (Open Systems Interconnection) model [Bellcore 98, 99; Telcordia 00] and the network/link layer [IETF 2005].
- Some examples of how a system can degrade through use (degradation) are given in [Nygard 18].
- 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). [Utas 05] has also written about escalating restart.

- 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 in 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].

Disaster recovery:

- A disaster is an event such as an earthquake, flood, or hurricane that destroys an entire data center. The U.S. National Institute of Standards and Technology (NIST) identifies eight different types of plans that should be considered in the event of a disaster, See Section 2.2 of NIST Special Publication 800-34, *Contingency Planning Guide for Federal Information Systems*, https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-34r1.pdf.

4.6 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) driverless car.

3. Write a concrete availability scenario for a program like Microsoft Word.

4. Redundancy is a key strategy for achieving high availability. Look at the patterns and 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 and deployability? How would you make a change to a system that is required to have 24/7 availability (i.e., no scheduled or unscheduled down time, ever)?

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

7. Which tactics are used by a load balancer (see Chapter 17) when it detects a failure of an instance?

8. Look up recovery point objective (RPO) and recovery time objective (RTO), and explain how these can be used to set a checkpoint interval when using the rollback tactic.