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

Availability	Downtime/90 Days	Downtime/Year
99.0%	21 hours, 36 minutes	3 days, 15.6 hours
99.9%	2 hours, 10 minutes	8 hours, 0 minutes, 46 seconds
99.99%	12 minutes, 58 seconds	52 minutes, 34 seconds
99.999%	1 minute, 18 seconds	5 minutes, 15 seconds
99.9999%	8 seconds	32 seconds

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.

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.

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

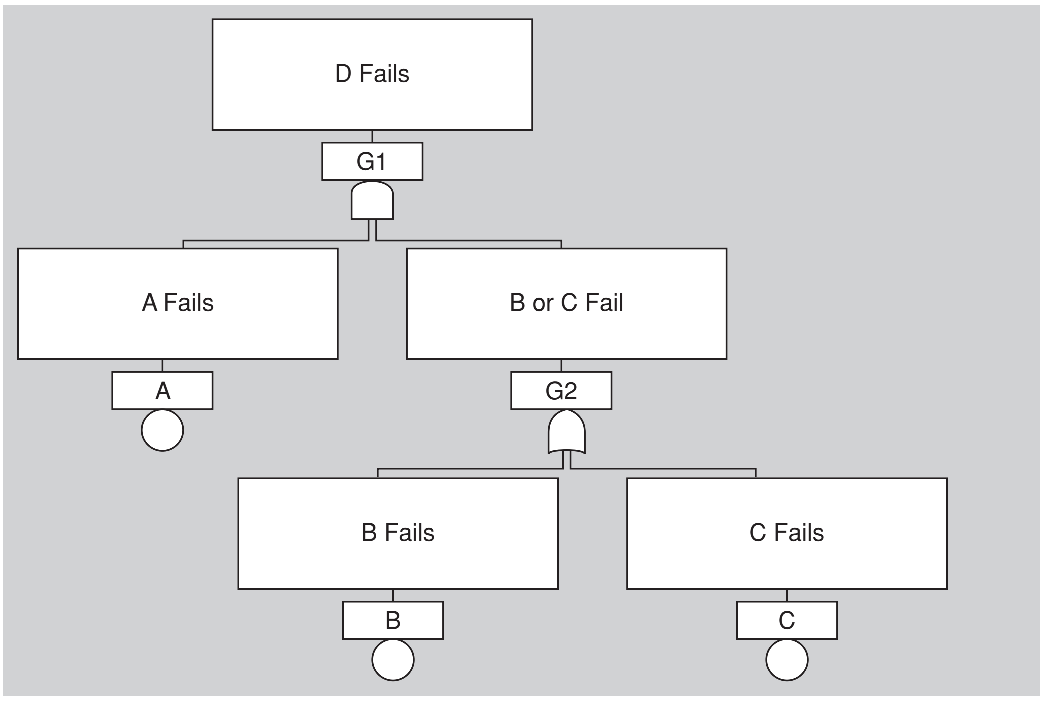
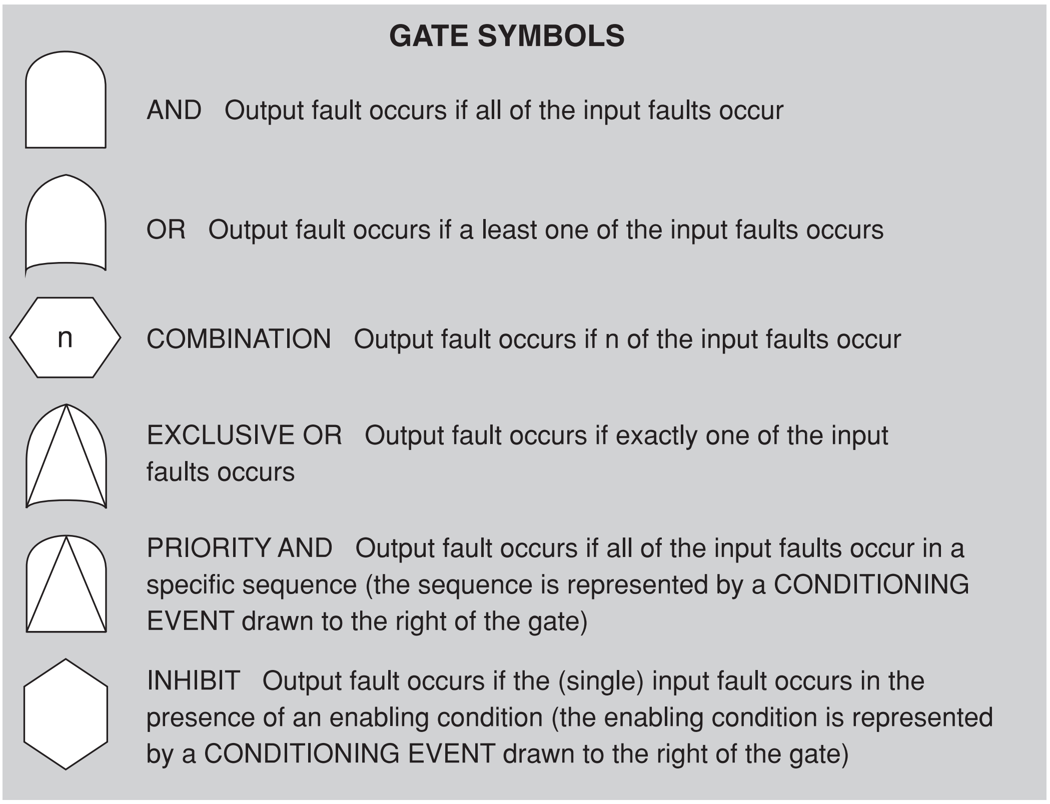


FIGURE 5.1 A simple fault tree. D fails if A fails and either B or C fails.

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").



GATE SYMBOLS

AND Output fault occurs if all of the input faults occur

OR Output fault occurs if a least one of the input faults occurs

COMBINATION Output fault occurs if n of the input faults occur

EXCLUSIVE OR Output fault occurs if exactly one of the input faults occurs

PRIORITY AND 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)

INHIBIT 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)

FIGURE 5.2 Fault tree gate symbols

TABLE 5.2 Failure Probabilities and Effects

				Effects	
Component	**Failure Probability**	**Failure Mode**	**% Failures by Mode**	**Critical**	**Noncritical**
		Open	90		X
A	1×10^{-3}	Short	5	X (5×10^{-5})	
		Other	5	X (5×10^{-5})	
B	1×10^{-3}	Open	90		X
		Short	5	X (5×10^{-5})	
		Other	5	X (5×10^{-5})	

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.

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 5.3 Availability General Scenario

Portion of Scenario	Possible Values
Source	Internal/external: people, hardware, software, physical infrastructure, physical environment
Stimulus	Fault: omission, crash, incorrect timing, incorrect response
Artifact	Processors, communication channels, persistent storage, processes
Environment	Normal operation, startup, shutdown, repair mode, degraded operation, overloaded operation
Response	Prevent the fault from becoming a failure Detect the fault: ▪ Log the fault ▪ Notify appropriate entities (people or systems) Recover from the fault: ▪ Disable source of events causing the fault ▪ Be temporarily unavailable while repair is being effected ▪ Fix or mask the fault/failure or contain the damage it causes ▪ Operate in a degraded mode while repair is being effected
Response Measure	Time or time interval when the system must be available Availability percentage (e.g., 99.999%) Time to detect the fault Time to repair the fault Time or time interval in which system can be in degraded mode 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

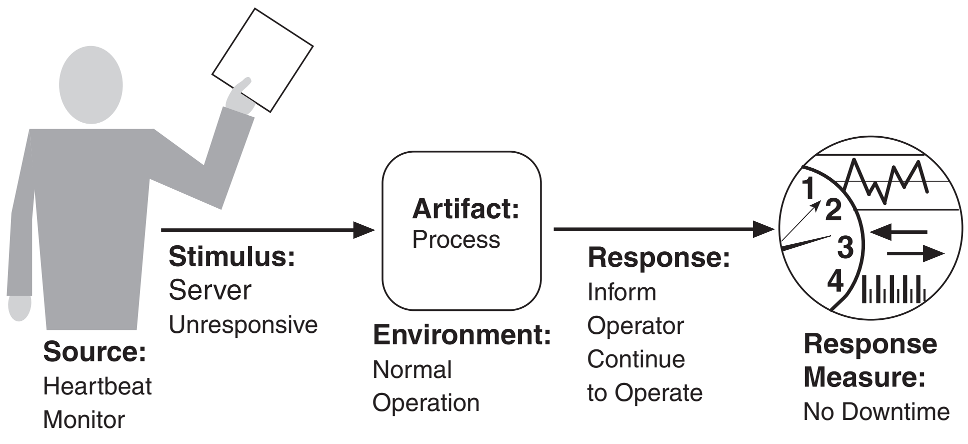


FIGURE 5.3 Sample concrete availability scenario

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.

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.

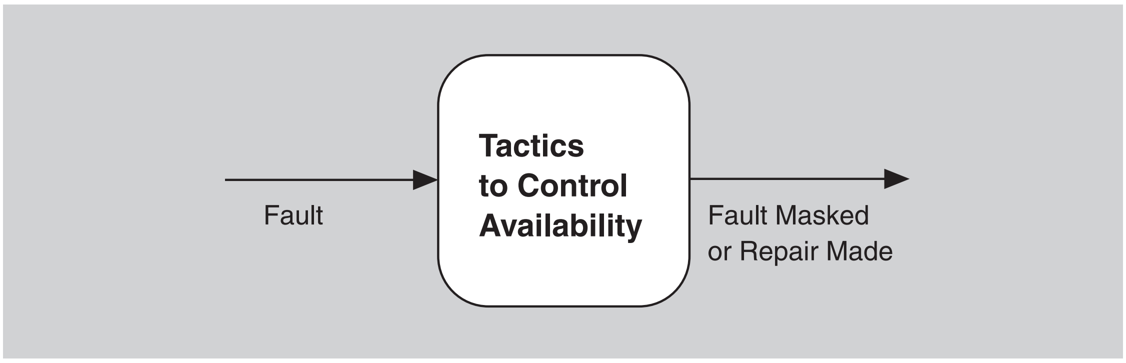


FIGURE 5.4 Goal of availability tactics

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 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

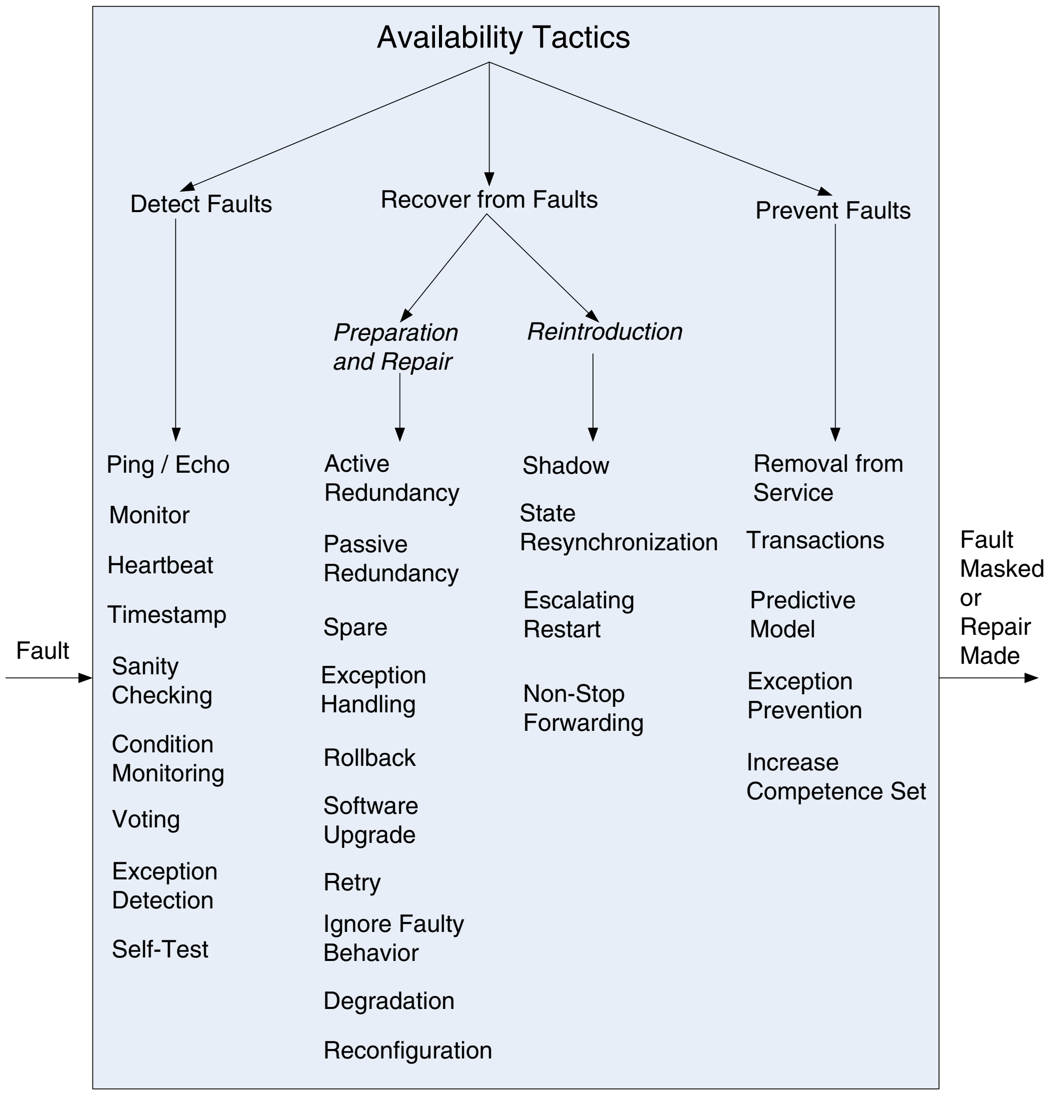


FIGURE 5.5 Availability tactics

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.

1. When the detection mechanism is implemented using a counter or timer that is periodically reset, this specialization of 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."

  - *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[2] 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

2. A protection group is a group of processing nodes where one or more nodes are "active," with the remaining nodes in the protection group serving as redundant spares.

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 fail-over 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

3. 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.

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

Category	Checklist
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.

Category	Checklist
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 ▪ How quickly the system can be reinstalled based on the units of delivery provided ▪ How to (re)assign runtime elements to processors, communication channels, and data stores ▪ When employing tactics that depend on redundancy of functionality, the mapping from modules to redundant components is important. For example, it is possible to write one module that contains code appropriate for both the active component and backup components in a protection group.
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.

continues

TABLE 5.4 Checklist to Support the Design and Analysis Process for Availability, *continued*

Category	**Checklist**
Binding Time	Determine how and when architectural elements are bound. If late binding is used to alternate between components that can themselves be sources of faults (e.g., processes, processors, communication channels), ensure the chosen availability strategy is sufficient to cover faults introduced by all sources. For example: ▪ If late binding is used to switch between artifacts such as processors that will receive or be the subject of faults, will the chosen fault detection and recovery mechanisms work for all possible bindings? ▪ If late binding is used to change the definition or tolerance of what constitutes a fault (e.g., how long a process can go without responding before a fault is assumed), is the recovery strategy chosen sufficient to handle all cases? For example, if a fault is flagged after 0.1 milliseconds, but the recovery mechanism takes 1.5 seconds to work, that might be an unacceptable mismatch. ▪ What are the availability characteristics of the late binding mechanism itself? Can it fail?
Choice of Technology	Determine the available technologies that can (help) detect faults, recover from faults, or reintroduce failed components. Determine what technologies are available that help the response to a fault (e.g., event loggers). Determine the availability characteristics of chosen technologies themselves: What faults can they recover from? What faults might they introduce into the system?

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 handlinghas 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?

8 Performance

An ounce of performance is worth pounds of promises.
—Mae West

It's about time.

Performance, that is: It's about time and the software system's ability to meet timing requirements. When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system or element's time-based response to those events is the essence is discussing performance.

Web-based system events come in the form of requests from users (numbering in the tens or tens of millions) via their clients such as web browsers. In a control system for an internal combustion engine, events come from the operator's controls and the passage of time; the system must control both the firing of the ignition when a cylinder is in the correct position and the mixture of the fuel to maximize power and efficiency and minimize pollution.

For a web-based system, the desired response might be expressed as number of transactions that can be processed in a minute. For the engine control system, the response might be the allowable variation in the firing time. In each case, the pattern of events arriving and the pattern of responses can be characterized, and this characterization forms the language with which to construct performance scenarios.

For much of the history of software engineering, performance has been the driving factor in system architecture. As such, it has frequently compromised the achievement of all other qualities. As the price/performance ratio of hardware continues to plummet and the cost of developing software continues to rise, other qualities have emerged as important competitors to performance.

Nevertheless, all systems have performance requirements, even if they are not expressed. For example, a word processing tool may not have any explicit performance requirement, but no doubt everyone would agree that waiting an hour (or a minute, or a second) before seeing a typed character appear on the screen is unacceptable. Performance continues to be a fundamentally important quality attribute for all software.

Performance is often linked to scalability—that is, increasing your system's capacity for work, while still performing well. Technically, scalability is making your system easy to change in a particular way, and so is a kind of modifiability. In addition, we address scalability explicitly in Chapter 12.

8.1 Performance General Scenario

A performance scenario begins with an event arriving at the system. Responding correctly to the event requires resources (including time) to be consumed. While this is happening, the system may be simultaneously servicing other events.

Concurrency

Concurrency is one of the more important concepts that an architect must understand and one of the least-taught in computer science courses. Concurrency refers to operations occurring in parallel. For example, suppose there is a thread that executes the statements

```
x := 1;
x++;
```

and another thread that executes the same statements. What is the value of *x* after both threads have executed those statements? It could be either 2 or 3. I leave it to you to figure out how the value 3 could occur—or should I say I interleave it to you?

Concurrency occurs any time your system creates a new thread, because threads, by definition, are independent sequences of control. Multitasking on your system is supported by independent threads. Multiple users are simultaneously supported on your system through the use of threads. Concurrency also occurs any time your system is executing on more than one processor, whether the processors are packaged separately or as multi-core processors. In addition, you must consider concurrency when parallel algorithms, parallelizing infrastructures such as map-reduce, or NoSQL databases are used by your system, or you utilize one of a variety of concurrent scheduling algorithms. In other words, concurrency is a tool available to you in many ways.

Concurrency, when you have multiple CPUs or wait states that can exploit it, is a good thing. Allowing operations to occur in parallel improves performance, because delays introduced in one thread allow the processor to progress on another thread. But because of the interleaving phenomenon just described (referred to as a *race condition*), concurrency must also be carefully managed by the architect.

As the example shows, race conditions can occur when there are two threads of control and there is shared state. The management of concurrency frequently comes down to managing how state is shared. One technique for preventing race conditions is to use locks to enforce sequential access to state. Another technique is to partition the state based on the thread executing a portion of code. That is, if there are two instances of *x* in our example, *x* is not shared by the two threads and there will not be a race condition.

Race conditions are one of the hardest types of bugs to discover; the occurrence of the bug is sporadic and depends on (possibly minute) differences in timing. I once had a race condition in an operating system that I could not track down. I put a test in the code so that the next time the race condition occurred, a debugging process was triggered. It took over a year for the bug to recur so that the cause could be determined.

Do not let the difficulties associated with concurrency dissuade you from utilizing this very important technique. Just use it with the knowledge that you must carefully identify critical sections in your code and ensure that race conditions will not occur in those sections.

—LB

Events can arrive in predictable patterns or mathematical distributions, or be unpredictable. An arrival pattern for events is characterized as *periodic*, *stochastic*, or *sporadic*:

- Periodic events arrive predictably at regular time intervals. For instance, an event may arrive every 10 milliseconds. Periodic event arrival is most often seen in real-time systems.
- Stochastic arrival means that events arrive according to some probabilistic distribution.
- Sporadic events arrive according to a pattern that is neither periodic nor stochastic. Even these can be characterized, however, in certain circumstances. For example, we might know that at most 600 events will occur in a minute, or that there will be at least 200 milliseconds between the arrival of any two events. (This might describe a system in which events correspond to keyboard strokes from a human user.) These are helpful characterizations, even though we don't know when any single event will arrive.

The response of the system to a stimulus can be measured by the following:

- *Latency.* The time between the arrival of the stimulus and the system's response to it.

- *Deadlines in processing.* In the engine controller, for example, the fuel should ignite when the cylinder is in a particular position, thus introducing a processing deadline.
- The *throughput* of the system, usually given as the number of transactions the system can process in a unit of time.
- The *jitter* of the response—the allowable variation in latency.
- The *number of events not processed* because the system was too busy to respond.

From these considerations we can now describe the individual portions of a general scenario for performance:

- *Source of stimulus.* The stimuli arrive either from external (possibly multiple) or internal sources.
- *Stimulus*. The stimuli are the event arrivals. The arrival pattern can be periodic, stochastic, or sporadic, characterized by numeric parameters.
- *Artifact*. The artifact is the system or one or more of its components.
- *Environment*. The system can be in various operational modes, such as normal, emergency, peak load, or overload.
- *Response*. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode).
- *Response measure.* The response measures are the time it takes to process the arriving events (latency or a deadline), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate).

The general scenario for performance is summarized in Table 8.1.

Figure 8.1 gives an example concrete performance scenario: Users initiate transactions under normal operations. The system processes the transactions with an average latency of two seconds.

TABLE 8.1 Performance General Scenario

Portion of Scenario	Possible Values
Source	Internal or external to the system
Stimulus	Arrival of a periodic, sporadic, or stochastic event
Artifact	System or one or more components in the system
Environment	Operational mode: normal, emergency, peak load, overload
Response	Process events, change level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate

8.2 Tactics for Performance

The goal of performance tactics is to generate a response to an event arriving at the system within some time-based constraint. The event can be single or a stream and is the trigger to perform computation. Performance tactics control the time within which a response is generated, as illustrated in Figure 8.2.

At any instant during the period after an event arrives but before the system's response t`o it is complete, either the system is working to respond to that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: processing time (when the system is working to respond) and blocked time (when the system is unable to respond).

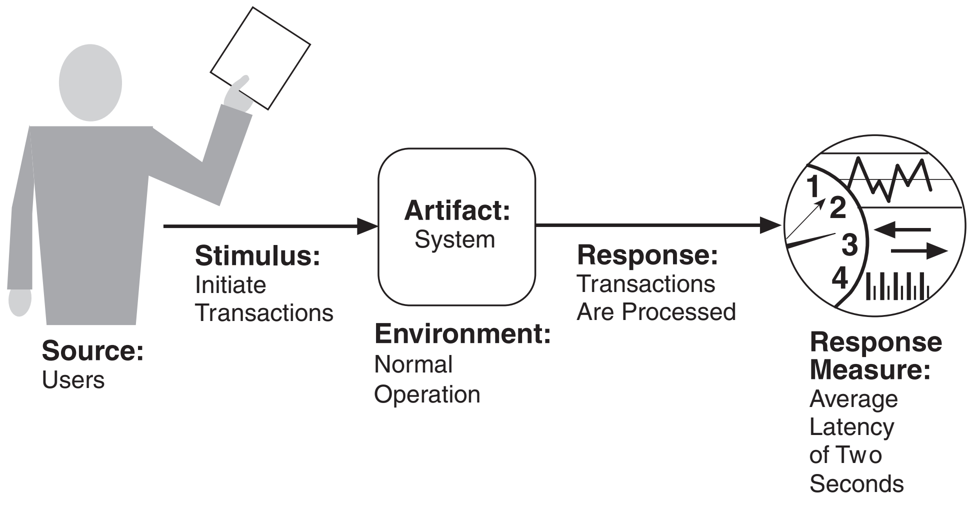


FIGURE 8.1 Sample concrete performance scenario

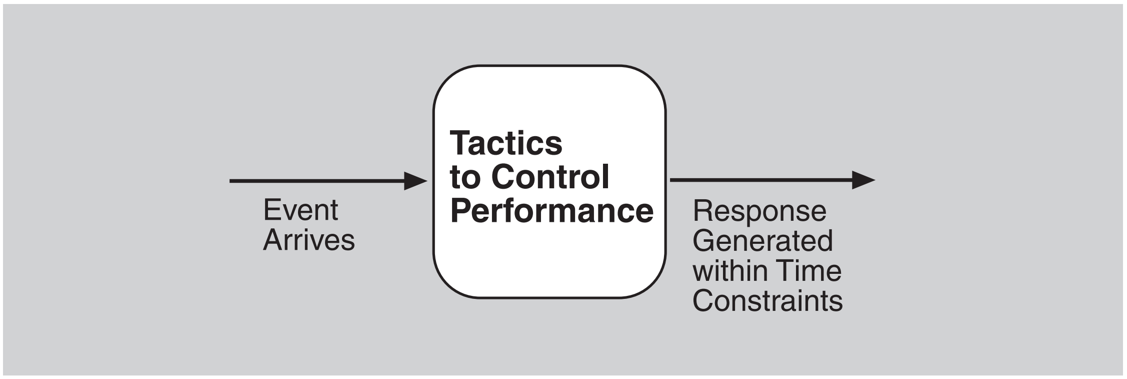

Event Arrives
Tactics to Control Performance
Response Generated within Time Constraints

FIGURE 8.2 The goal of performance tactics

- *Processing time.* Processing consumes resources, which takes time. Events are handled by the execution of one or more components, whose time expended is a resource. Hardware resources include CPU, data stores, network communication bandwidth, and memory. Software resources include entities defined by the system under design. For example, buffers must be managed and access to critical sections[1] must be made sequential.

  For example, suppose a message is generated by one component. It might be placed on the network, after which it arrives at another component. It is then placed in a buffer; transformed in some fashion; processed according to some algorithm; transformed for output; placed in an output buffer; and sent onward to another component, another system, or some actor. Each of these steps consumes resources and time and contributes to the overall latency of the processing of that event.

  Different resources behave differently as their utilization approaches their capacity—that is, as they become saturated. For example, as a CPU becomes more heavily loaded, performance usually degrades fairly steadily. On the other hand, when you start to run out of memory, at some point the page swapping becomes overwhelming and performance crashes suddenly.

- *Blocked time.* A computation can be blocked because of contention for some needed resource, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available:

  - *Contention for resources.* Many resources can only be used by a single client at a time. This means that other clients must wait for access to those resources. Figure 8.2 shows events arriving at the system. These events may be in a single stream or in multiple streams. Multiple streams vying for the same resource or different events in the same stream vying for the same resource contribute to latency. The more contention for a resource, the more likelihood of latency being introduced.
  - *Availability of resources.* Even in the absence of contention, computation cannot proceed if a resource is unavailable. Unavailability may be caused by the resource being offline or by failure of the component or for some other reason. In any case, you must identify places where resource unavailability might cause a significant contribution to overall latency. Some of our tactics are intended to deal with this situation.
  - *Dependency on other computation.* A computation may have to wait because it must synchronize with the results of another computation or because it is waiting for the results of a computation that it initiated. If a component calls another component and must wait for that component to respond, the time can be significant if the called component is at the other end of a network (as opposed to co-located on the same processor).

1. A critical section is a section of code in a multi-threaded system in which at most one thread may be active at any time.

With this background, we turn to our tactic categories. We can either reduce demand for resources or make the resources we have handle the demand more effectively:

- *Control resource demand.* This tactic operates on the demand side to produce smaller demand on the resources that will have to service the events.
- *Manage resources.* This tactic operates on the response side to make the resources at hand work more effectively in handling the demands put to them.

Control Resource Demand

One way to increase performance is to carefully manage the demand for resources. This can be done by reducing the number of events processed by enforcing a sampling rate, or by limiting the rate at which the system responds to events. In addition, there are a number of techniques for ensuring that the resources that you do have are applied judiciously:

- *Manage sampling rate*. If it is possible to reduce the sampling frequency at which a stream of environmental data is captured, then demand can be reduced, typically with some attendant loss of fidelity. This is common in signal processing systems where, for example, different codecs can be chosen with different sampling rates and data formats. This design choice is made to maintain predictable levels of latency; you must decide whether having a lower fidelity but consistent stream of data is preferable to losing packets of data.
- *Limit event response*. When discrete events arrive at the system (or element) too rapidly to be processed, then the events must be queued until they can be processed. Because these events are discrete, it is typically not desirable to "downsample" them. In such a case, you may choose to process events only up to a set maximum rate, thereby ensuring more predictable processing when the events are actually processed. This tactic could be triggered by a queue size or processor utilization measure exceeding some warning level. If you adopt this tactic and it is unacceptable to lose any events, then you must ensure that your queues are large enough to handle the worst case. If, on the other hand, you choose to drop events, then you need to choose a policy for handling this situation: Do you log the dropped events, or simply ignore them? Do you notify other systems, users, or administrators?
- *Prioritize events*. If not all events are equally important, you can impose a priority scheme that ranks events according to how important it is to service them. If there are not enough resources available to service them when they arise, low-priority events might be ignored. Ignoring events consumes minimal resources (including time), and thus increases performance compared to a system that services all events all the time. For example, a building

management system may raise a variety of alarms. Life-threatening alarms such as a fire alarm should be given higher priority than informational alarms such as a room is too cold.

- *Reduce overhead*. The use of intermediaries (so important for modifiability, as we saw in Chapter 7) increases the resources consumed in processing an event stream, and so removing them improves latency. This is a classic modifiability/performance tradeoff. Separation of concerns, another linchpin of modifiability, can also increase the processing overhead necessary to service an event if it leads to an event being serviced by a chain of components rather than a single component. The context switching and intercomponent communication costs add up, especially when the components are on different nodes on a network. A strategy for reducing computational overhead is to co-locate resources. Co-location may mean hosting cooperating components on the same processor to avoid the time delay of network communication; it may mean putting the resources in the same runtime software component to avoid even the expense of a subroutine call. A special case of reducing computational overhead is to perform a periodic cleanup of resources that have become inefficient. For example, hash tables and virtual memory maps may require recalculation and reinitialization. Another common strategy is to execute single-threaded servers (for simplicity and avoiding contention) and split workload across them.
- *Bound execution times*. Place a limit on how much execution time is used to respond to an event. For iterative, data-dependent algorithms, limiting the number of iterations is a method for bounding execution times. The cost is usually a less accurate computation. If you adopt this tactic, you will need to assess its effect on accuracy and see if the result is "good enough." This resource management tactic is frequently paired with the manage sampling rate tactic.
- *Increase resource efficiency.* Improving the algorithms used in critical areas will decrease latency.

Manage Resources

Even if the demand for resources is not controllable, the management of these resources can be. Sometimes one resource can be traded for another. For example, intermediate data may be kept in a cache or it may be regenerated depending on time and space resource availability. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk. Here are some resource management tactics:

- *Increase resources.* Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

Cost is usually a consideration in the choice of resources, but increasing the resources is definitely a tactic to reduce latency and in many cases is the cheapest way to get immediate improvement.

- *Introduce concurrency*. If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities. Once concurrency has been introduced, scheduling policies can be used to achieve the goals you find desirable. Different scheduling policies may maximize fairness (all requests get equal time), throughput (shortest time to finish first), or other goals. (See the sidebar.)
- *Maintain multiple copies of computations*. Multiple servers in a client-server pattern are replicas of computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a single server. A *load balancer* is a piece of software that assigns new work to one of the available duplicate servers; criteria for assignment vary but can be as simple as round-robin or assigning the next request to the least busy server.
- *Maintain multiple copies of data. Caching* is a tactic that involves keeping copies of data (possibly one a subset of the other) on storage with different access speeds. The different access speeds may be inherent (memory versus secondary storage) or may be due to the necessity for network communication. *Data replication* involves keeping separate copies of the data to reduce the contention from multiple simultaneous accesses. Because the data being cached or replicated is usually a copy of existing data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume. Another responsibility is to choose the data to be cached. Some caches operate by merely keeping copies of whatever was recently requested, but it is also possible to predict users' future requests based on patterns of behavior, and begin the calculations or prefetches necessary to comply with those requests before the user has made them.
- *Bound queue sizes*. This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals. If you adopt this tactic, you need to adopt a policy for what happens when the queues overflow and decide if not responding to lost events is acceptable. This tactic is frequently paired with the limit event response tactic.
- *Schedule resources*. Whenever there is contention for a resource, the resource must be scheduled. Processors are scheduled, buffers are scheduled, and networks are scheduled. Your goal is to understand the characteristics of each resource's use and choose the scheduling strategy that is compatible with it. (See the sidebar.)

The tactics for performance are summarized in Figure 8.3.

Scheduling Policies

A *scheduling policy* conceptually has two parts: a priority assignment and dispatching. All scheduling policies assign priorities. In some cases the assignment is as simple as first-in/first-out (or FIFO). In other cases, it can be tied to the deadline of the request or its semantic importance. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, maximizing throughput, preventing starvation to ensure fairness, and so forth. You need to be aware of these possibly conflicting criteria and the effect that the chosen tactic has on meeting them.

A high-priority event stream can be dispatched only if the resource to which it is being assigned is available. Sometimes this depends on preempting the current user of the resource. Possible preemption options are as follows: can occur anytime, can occur only at specific preemption points, and executing processes cannot be preempted. Some common scheduling policies are these:

- *First-in/first-out.* FIFO queues treat all requests for resources as equals and satisfy them in turn. One possibility with a FIFO queue is that one request will be stuck behind another one that takes a long time to generate a response. As long as all of the requests are truly equal, this is not a problem, but if some requests are of higher priority than others, it is problematic.
- *Fixed-priority scheduling.* Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order. This strategy ensures better service for higher priority requests. But it admits the possibility of a lower priority, but important, request taking an arbitrarily long time to be serviced, because it is stuck behind a series of higher priority requests. Three common prioritization strategies are these:
  - *Semantic importance.* Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
  - *Deadline monotonic.* Deadline monotonic. Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines. This scheduling policy is used when streams of different priorities with real-time deadlines are to be scheduled.
  - *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods. This scheduling policy is a special case of deadline monotonic but is better known and more likely to be supported by the operating system.
- *Dynamic priority scheduling.* Strategies include these:
  - *Round-robin.* Round-robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order. A special form of

round-robin is a cyclic executive, where assignment possibilities are at fixed time intervals.
- *Earliest-deadline-first.* Earliest-deadline-first. Earliest-deadline-first assigns priorities based on the pending requests with the earliest deadline.
- *Least-slack-first.* This strategy assigns the highest priority to the job having the least "slack time," which is the difference between the execution time remaining and the time to the job's deadline.

For a single processor and processes that are preemptible (that is, it is possible to suspend processing of one task in order to service a task whose deadline is drawing near), both the earliest-deadline and least-slack scheduling strategies are optimal. That is, if the set of processes can be scheduled so that all deadlines are met, then these strategies will be able to schedule that set successfully.

- *Static scheduling.* A cyclic executive schedule is a scheduling strategy where the preemption points and the sequence of assignment to the resource are determined offline. The runtime overhead of a scheduler is thereby obviated.

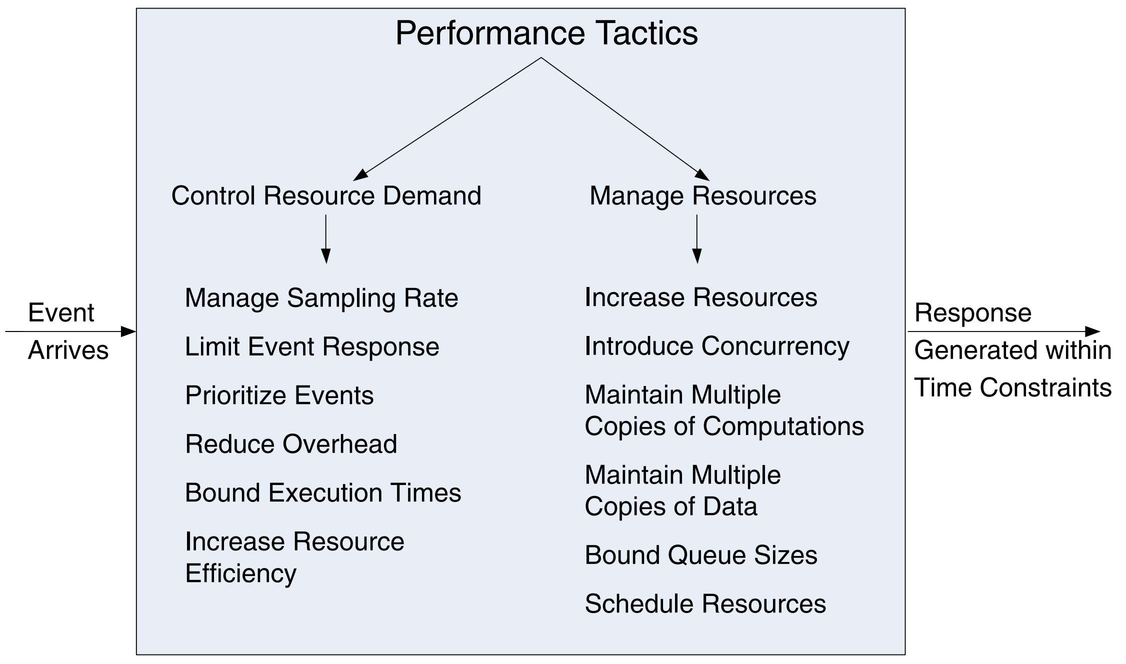


FIGURE 8.3 Performance tactics

Performance Tactics on the Road

Tactics are generic design principles. To exercise this point, think about the design of the systems of roads and highways where you live. Traffic engineers employ a bunch of design "tricks" to optimize the performance of these complex systems, where performance has a number of measures, such as throughput (how many cars per hour get from the suburbs to the football stadium), average-case latency (how long it takes, on average, to get from your house to downtown), and worst-case latency (how long does it take an emergency vehicle to get you to the hospital). What are these tricks? None other than our good old buddies, tactics.

Let's consider some examples:

- *Manage event rate.* Lights on highway entrance ramps let cars onto the highway only at set intervals, and cars must wait (queue) on the ramp for their turn.
- *Prioritize events.* Ambulances and police, with their lights and sirens going, have higher priority than ordinary citizens; some highways have high-occupancy vehicle (HOV) lanes, giving priority to vehicles with two or more occupants.
- *Maintain multiple copies.* Add traffic lanes to existing roads, or build parallel routes.

In addition, there are some tricks that users of the system can employ:

- *Increase resources.* Buy a Ferrari, for example. All other things being equal, the fastest car with a competent driver on an open road will get you to your destination more quickly.
- *Increase efficiency.* Find a new route that is quicker and/or shorter than your current route.
- *Reduce computational overhead.* You can drive closer to the car in front of you, or you can load more people into the same vehicle (that is, carpooling).

What is the point of this discussion? To paraphrase Gertrude Stein: performance is performance is performance. Engineers have been analyzing and optimizing systems for centuries, trying to improve their performance, and they have been employing the same design strategies to do so. So you should feel some comfort in knowing that when you try to improve the performance of your computer-based system, you are applying tactics that have been thoroughly "road tested."

—RK

8.3 A Design Checklist for Performance

Table 8.2 is a checklist to support the design and analysis process for performance.

TABLE 8.2 Checklist to Support the Design and Analysis Process for Performance

Category	Checklist
Allocation of Responsibilities	Determine the system's responsibilities that will involve heavy loading, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those responsibilities, identify the processing requirements of each responsibility, and determine whether they may cause bottlenecks. Also, identify additional responsibilities to recognize and process requests appropriately, including ▪ Responsibilities that result from a thread of control crossing process or processor boundaries ▪ Responsibilities to manage the threads of control—allocation and deallocation of threads, maintaining thread pools, and so forth ▪ Responsibilities for scheduling shared resources or managing performance-related artifacts such as queues, buffers, and caches For the responsibilities and resources you identified, ensure that the required performance response can be met (perhaps by building a performance model to help in the evaluation).
Coordination Model	Determine the elements of the system that must coordinate with each other—directly or indirectly—and choose communication and coordination mechanisms that do the following: ▪ Support any introduced concurrency (for example, is it thread safe?), event prioritization, or scheduling strategy ▪ Ensure that the required performance response can be delivered ▪ Can capture periodic, stochastic, or sporadic event arrivals, as needed ▪ Have the appropriate properties of the communication mechanisms; for example, stateful, stateless, synchronous, asynchronous, guaranteed delivery, throughput, or latency
Data Model	Determine those portions of the data model that will be heavily loaded, have time-critical response requirements, are heavily used, or impact portions of the system where heavy loads or time-critical events occur. For those data abstractions, determine the following: ▪ Whether maintaining multiple copies of key data would benefit performance ▪ Whether partitioning data would benefit performance ▪ Whether reducing the processing requirements for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is possible ▪ Whether adding resources to reduce bottlenecks for the creation, initialization, persistence, manipulation, translation, or destruction of the enumerated data abstractions is feasible

continues

TABLE 8.2 Checklist to Support the Design and Analysis Process for Performance, *continued*

Category	Checklist
Mapping among Architectural Elements	Where heavy network loading will occur, determine whether co-locating some components will reduce loading and improve overall efficiency. Ensure that components with heavy computation requirements are assigned to processors with the most processing capacity. Determine where introducing concurrency (that is, allocating a piece of functionality to two or more copies of a component running simultaneously) is feasible and has a significant positive effect on performance. Determine whether the choice of threads of control and their associated responsibilities introduces bottlenecks.
Resource Management	Determine which resources in your system are critical for performance. For these resources, ensure that they will be monitored and managed under normal and overloaded system operation. For example: ▪ System elements that need to be aware of, and manage, time and other performance-critical resources ▪ Process/thread models ▪ Prioritization of resources and access to resources ▪ Scheduling and locking strategies ▪ Deploying additional resources on demand to meet increased loads
Binding Time	For each element that will be bound after compile time, determine the following: ▪ Time necessary to complete the binding ▪ Additional overhead introduced by using the late binding mechanism Ensure that these values do not pose unacceptable performance penalties on the system.
Choice of Technology	Will your choice of technology let you set and meet hard, real-time deadlines? Do you know its characteristics under load and its limits? Does your choice of technology give you the ability to set the following: ▪ Scheduling policy ▪ Priorities ▪ Policies for reducing demand ▪ Allocation of portions of the technology to processors ▪ Other performance-related parameters Does your choice of technology introduce excessive overhead for heavily used operations?

8.4 Summary

Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behavior. Performance can be measured in terms of throughput and latency for both interactive and embedded real-time systems, although throughput is usually more important in interactive systems, and latency is more important in embedded systems.

Performance can be improved by reducing demand or by managing resources more appropriately. Reducing demand will have the side effect of reducing fidelity or refusing to service some requests. Managing resources more appropriately can be done through scheduling, replication, or just increasing the resources available.

8.5 For Further Reading

Performance has a rich body of literature. Here are some books we recommend:

- *Software Performance and Scalability: A Quantitative Approach* [Liu 09]. This books covers performance geared toward enterprise applications, with an emphasis on queuing theory and measurement.
- *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software* [Smith 01]. This book covers designing with performance in mind, with emphasis on building (and populating with real data) practical predictive performance models.
- *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems* [Douglass 99].
- *Real-Time Systems* [Liu 00].
- *Pattern-Oriented Software Architecture Volume 3: Patterns for Resource Management* [Kircher 03].

8.6 Discussion Questions

1. “Every system has real-time performance constraints.” Discuss. Or provide a counterexample.
2. Write a performance scenario that describes the average on-time flight arrival performance for an airline.

3. Write several performance scenarios for an automatic teller machine. Think about whether your major concern is worst-case latency, average-case latency, throughput, or some other response measure. How would you modify your automatic teller machine design to accommodate these scenarios?

4. Web-based systems often use *proxy servers*, which are the first element of the system to receive a request from a client (such as your browser). Proxy servers are able to serve up often-requested web pages, such as a company's home page, without bothering the real application servers that carry out transactions. There may be many proxy servers, and they are often located geographically close to large user communities, to decrease response time for routine requests. What performance tactics do you see at work here?

5. A fundamental difference between coordination mechanisms is whether interaction is synchronous or asynchronous. Discuss the advantages and disadvantages of each with respect to each of the performance responses: latency, deadline, throughput, jitter, miss rate, data loss, or any other required performance-related response you may be used to.

6. Find real-world (that is, nonsoftware) examples of applying each of the manage-resources tactics. For example, suppose you were managing a brick-and-mortar big-box retail store. How would you get people through the checkout lines faster using these tactics?

7. User interface frameworks typically are single-threaded. Why is this so and what are the performance implications of this single-threading?

9 Security

With Jungwoo Ryoo and Phil Laplante

Your personal identity isn't worth quite as much as it used to be—at least to thieves willing to swipe it. According to experts who monitor such markets, the value of stolen credit card data may range from $3 to as little as 40 cents. That's down tenfold from a decade ago—even though the cost to an individual who has a credit card stolen can soar into the hundreds of dollars.
—Forbes.com (Taylor Buley. "Hackonomics," Forbes.com, October 27, 2008, www.forbes.com/2008/10/25/credit-card-theft-tech-security-cz_tb1024theft.html)

Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized. An action taken against a computer system with the intention of doing harm is called an attack and can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

The simplest approach to characterizing security has three characteristics: confidentiality, integrity, and availability (CIA):

1. *Confidentiality* is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
2. *Integrity* is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
3. *Availability* is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering book from an online bookstore.

Other characteristics that are used to support CIA are these:

4. *Authentication* verifies the identities of the parties to a transaction and checks if they are truly who they claim to be. For example, when you get an email purporting to come from a bank, authentication guarantees that it actually comes from the bank.
5. *Nonrepudiation* guarantees that the sender of a message cannot later deny having sent the message, and that the recipient cannot deny having received the message. For example, you cannot deny ordering something from the Internet, or the merchant cannot disclaim getting your order.
6. *Authorization* grants a user the privileges to perform a task. For example, an online banking system authorizes a legitimate user to access his account.

We will use these characteristics in our general scenarios for security. Approaches to achieving security can be characterized as those that detect attacks, those that resist attacks, those that react to attacks, and those that recover from successful attacks. The objects that are being protected from attacks are data at rest, data in transit, and computational processes.

9.1 Security General Scenario

One technique that is used in the security domain is threat modeling. An "attack tree," similar to a fault tree discussed in Chapter 5, is used by security engineers to determine possible threats. The root is a successful attack and the nodes are possible direct causes of that successful attack. Children nodes decompose the direct causes, and so forth. An attack is an attempt to break CIA, and the leaves of attack trees would be the stimulus in the scenario. The response to the attack is to preserve CIA or deter attackers through monitoring of their activities. From these considerations we can now describe the individual portions of a security general scenario. These are summarized in Table 9.1, and an example security scenario is given in Figure 9.1.

- *Source of stimulus*. The source of the attack may be either a human or another system. It may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
- *Stimulus.* The stimulus is an attack. We characterize this as an unauthorized attempt to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
- *Artifact*. The target of the attack can be either the services of the system, the data within it, or the data produced or consumed by the system. Some attacks are made on particular components of the system known to be vulnerable.

- *Environment*. The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to a network, fully operational, partially operational, or not operational.
- *Response.* The system should ensure that transactions are carried out in a fashion such that data or services are protected from unauthorized access; data or services are not being manipulated without authorization; parties to a transaction are identified with assurance; the parties to the transaction cannot repudiate their involvements; and the data, resources, and system services will be available for legitimate use.

  The system should also track activities within it by recording access or modification; attempts to access data, resources, or services; and notifying appropriate entities (people or systems) when an apparent attack is occurring.
- *Response measure*. Measures of a system's response include how much of a system is compromised when a particular component or data value is compromised, how much time passed before an attack was detected, how many attacks were resisted, how long it took to recover from a successful attack, and how much data was vulnerable to a particular attack.

Table 9.1 enumerates the elements of the general scenario, which characterize security, and Figure 9.1 shows a sample concrete scenario: A disgruntled employee from a remote location attempts to modify the pay rate table during normal operations. The system maintains an audit trail, and the correct data is restored within a day.

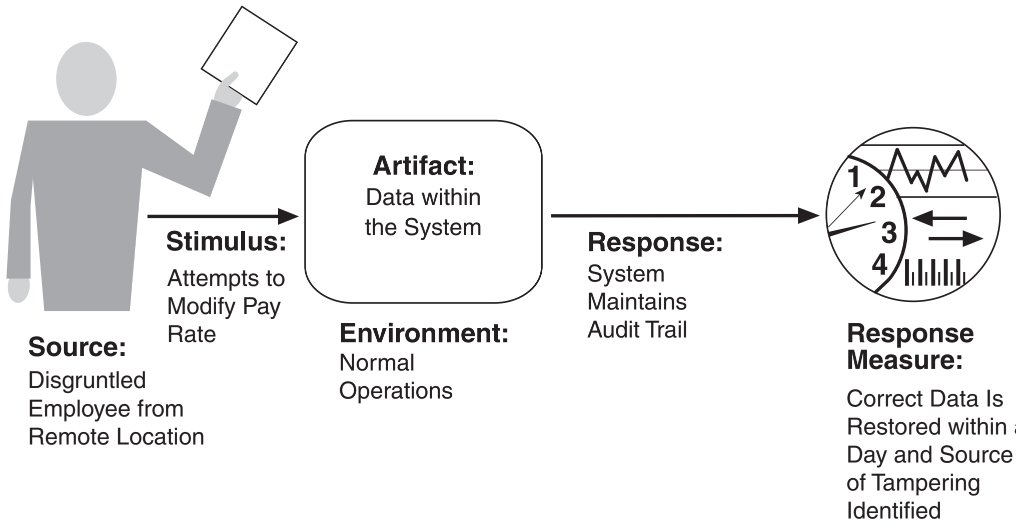

Artifact:
Data within the System
Stimulus:
Attempts to Modify Pay Rate
Source:
Disgruntled Employee from Remote Location
Environment:
Normal Operations
Response:
System Maintains Audit Trail
1
2
3
4
Response Measure:
Correct Data Is Restored within a Day and Source of Tampering Identified

FIGURE 9.1 Sample concrete security scenario

TABLE 9.1 Security General Scenario

Portion of Scenario	**Possible Values**
Source	Human or another system which may have been previously identified (either correctly or incorrectly) or may be currently unknown. A human attacker may be from outside the organization or from inside the organization.
Stimulus	Unauthorized attempt is made to display data, change or delete data, access system services, change the system's behavior, or reduce availability.
Artifact	System services, data within the system, a component or resources of the system, data produced or consumed by the system
Environment	The system is either online or offline; either connected to or disconnected from a network; either behind a firewall or open to a network; fully operational, partially operational, or not operational.
Response	Transactions are carried out in a fashion such that ▪ Data or services are protected from unauthorized access. ▪ Data or services are not being manipulated without authorization. ▪ Parties to a transaction are identified with assurance. ▪ The parties to the transaction cannot repudiate their involvements. ▪ The data, resources, and system services will be available for legitimate use. The system tracks activities within it by ▪ Recording access or modification ▪ Recording attempts to access data, resources, or services ▪ Notifying appropriate entities (people or systems) when an apparent attack is occurring
Response Measure	One or more of the following: ▪ How much of a system is compromised when a particular component or data value is compromised ▪ How much time passed before an attack was detected ▪ How many attacks were resisted ▪ How long does it take to recover from a successful attack ▪ How much data is vulnerable to a particular attack

9.2 Tactics for Security

One method for thinking about how to achieve security in a system is to think about physical security. Secure installations have limited access (e.g., by using security checkpoints), have means of detecting intruders (e.g., by requiring legitimate visitors to wear badges), have deterrence mechanisms such as armed guards, have reaction mechanisms such as automatic locking of doors, and have recovery mechanisms such as off-site backup. These lead to our four categories of tactics: detect, resist, react, and recover. Figure 9.2 shows these categories as the goal of security tactics.

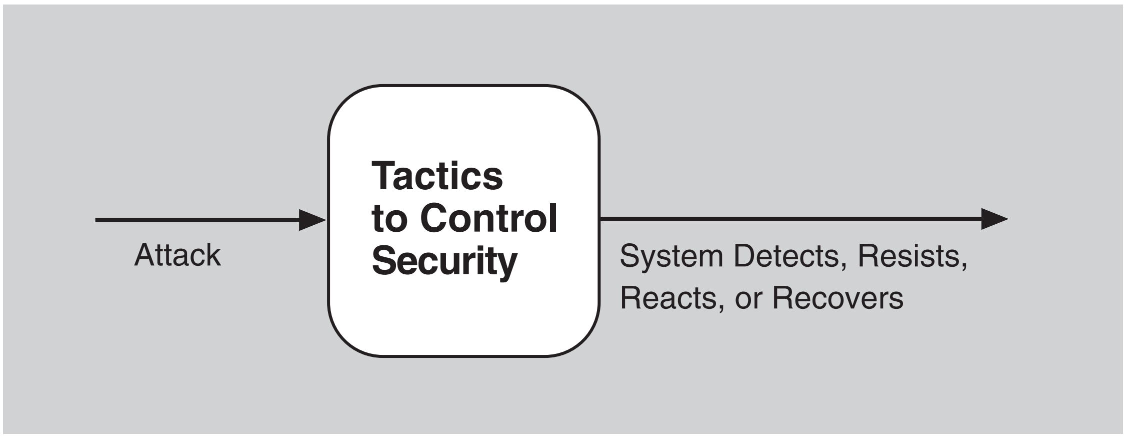

Attack
Tactics to Control Security
System Detects, Resists, Reacts, or Recovers

FIGURE 9.2 The goal of security tactics

Detect Attacks

The detect attacks category consists of four tactics: detect intrusion, detect service denial, verify message integrity, and detect message delay.

- *Detect intrusion* is the comparison of network traffic or service request patterns *within* a system to a set of signatures or known patterns of malicious behavior stored in a database. The signatures can be based on protocol, TCP flags, payload sizes, applications, source or destination address, or port number.
- *Detect service denial* is the comparison of the pattern or signature of network traffic *coming into* a system to historic profiles of known denial-of-service attacks.
- *Verify message integrity.* This tactic employs techniques such as checksums or hash values to verify the integrity of messages, resource files, deployment files, and configuration files. A checksum is a validation mechanism wherein the system maintains redundant information for configuration files and messages, and uses this redundant information to verify the configuration file or message when it is used. A hash value is a unique string generated by a hashing function whose input could be configuration files or messages. Even a slight change in the original files or messages results in a significant change in the hash value.
- *Detect message delay* is intended to detect potential man-in-the-middle attacks, where a malicious party is intercepting (and possibly modifying) messages. By checking the time that it takes to deliver a message, it is possible to detect suspicious timing behavior, where the time it takes to deliver a message is highly variable.

Resist Attacks

There are a number of well-known means of resisting an attack:

- *Identify actors*. Identifying "actors" is really about identifying the source of any external input to the system. Users are typically identified through user IDs. Other systems may be "identified" through access codes, IP addresses, protocols, ports, and so on.
- *Authenticate actors*. Authentication means ensuring that an actor (a user or a remote computer) is actually who or what it purports to be. Passwords, one-time passwords, digital certificates, and biometric identification provide a means for authentication.
- *Authorize actors*. Authorization means ensuring that an authenticated actor has the rights to access and modify either data or services. This mechanism is usually enabled by providing some access control mechanisms within a system. Access control can be by an actor or by an actor class. Classes of actors can be defined by actor groups, by actor roles, or by lists of individuals.
- *Limit access*. Limiting access involves controlling what and who may access which parts of a system. This may include limiting access to resources such as processors, memory, and network connections, which may be achieved by using process management, memory protection, blocking a host, closing a port, or rejecting a protocol. For example, a firewall is a single point of access to an organization's intranet. A demilitarized zone (DMZ) is a subnet between the Internet and an intranet, protected by two firewalls: one facing the Internet and the other the intranet. A DMZ is used when an organization wants to let external users access services that should be publicly available outside the intranet. This way the number of open ports in the internal firewall can be minimized. This tactic also limits access for actors (by identifying, authenticating, and authorizing them).
- *Limit exposure.* Limiting exposure refers to ultimately and indirectly reducing the probability of a successful attack, or restricting the amount of potential damage. This can be achieved by concealing facts about a system to be protected ("security by obscurity") or by dividing and distributing critical resources so that the exploitation of a single weakness cannot fully compromise any resource ("don't put all your eggs in one basket"). For example, a design decision to hide how many entry points a system has is a way of limiting exposure. A decision to distribute servers amongst several geographically dispersed data centers is also a way of limiting exposure.
- *Encrypt data*. Data should be protected from unauthorized access. Confidentiality is usually achieved by applying some form of encryption to data and to communication. Encryption provides extra protection to persistently maintained data beyond that available from authorization. Communication links, on the other hand, may not have authorization controls. In such cases, encryption is the only protection for passing data over publicly accessible communication links. The link can be implemented by a virtual private network (VPN) or by a Secure Sockets Layer (SSL) for

a web-based link. Encryption can be symmetric (both parties use the same key) or asymmetric (public and private keys).

- *Separate entities*. Separating different entities within the system can be done through physical separation on different servers that are attached to different networks; the use of virtual machines (see Chapter 26 for a discussion of virtual machines); or an "air gap," that is, by having no connection between different portions of a system. Finally, sensitive data is frequently separated from nonsensitive data to reduce the attack possibilities from those who have access to nonsensitive data.
- *Change default settings*. Many systems have default settings assigned when the system is delivered. Forcing the user to change those settings will prevent attackers from gaining access to the system through settings that are, generally, publicly available.

React to Attacks

Several tactics are intended to respond to a potential attack:

- *Revoke access.* If the system or a system administrator believes that an attack is underway, then access can be severely limited to sensitive resources, even for normally legitimate users and uses. For example, if your desktop has been compromised by a virus, your access to certain resources may be limited until the virus is removed from your system.
- *Lock computer*. Repeated failed login attempts may indicate a potential attack. Many systems limit access from a particular computer if there are repeated failed attempts to access an account from that computer. Legitimate users may make mistakes in attempting to log in. Therefore, the limited access may only be for a certain time period.
- *Inform actors*. Ongoing attacks may require action by operators, other personnel, or cooperating systems. Such personnel or systems—the set of relevant actors—must be notified when the system has detected an attack.

Recover from Attacks

Once a system has detected and attempted to resist an attack, it needs to recover. Part of recovery is restoration of services. For example, additional servers or network connections may be kept in reserve for such a purpose. Since a successful attack can be considered a kind of failure, the set of availability tactics (from Chapter 5) that deal with recovering from a failure can be brought to bear for this aspect of security as well.

In addition to the availability tactics that permit restoration of services, we need to maintain an audit trail. We audit—that is, keep a record of user and system actions and their effects—to help trace the actions of, and to identify, an attacker. We may analyze audit trails to attempt to prosecute attackers, or to create better defenses in the future.

The set of security tactics is shown in Figure 9.3.

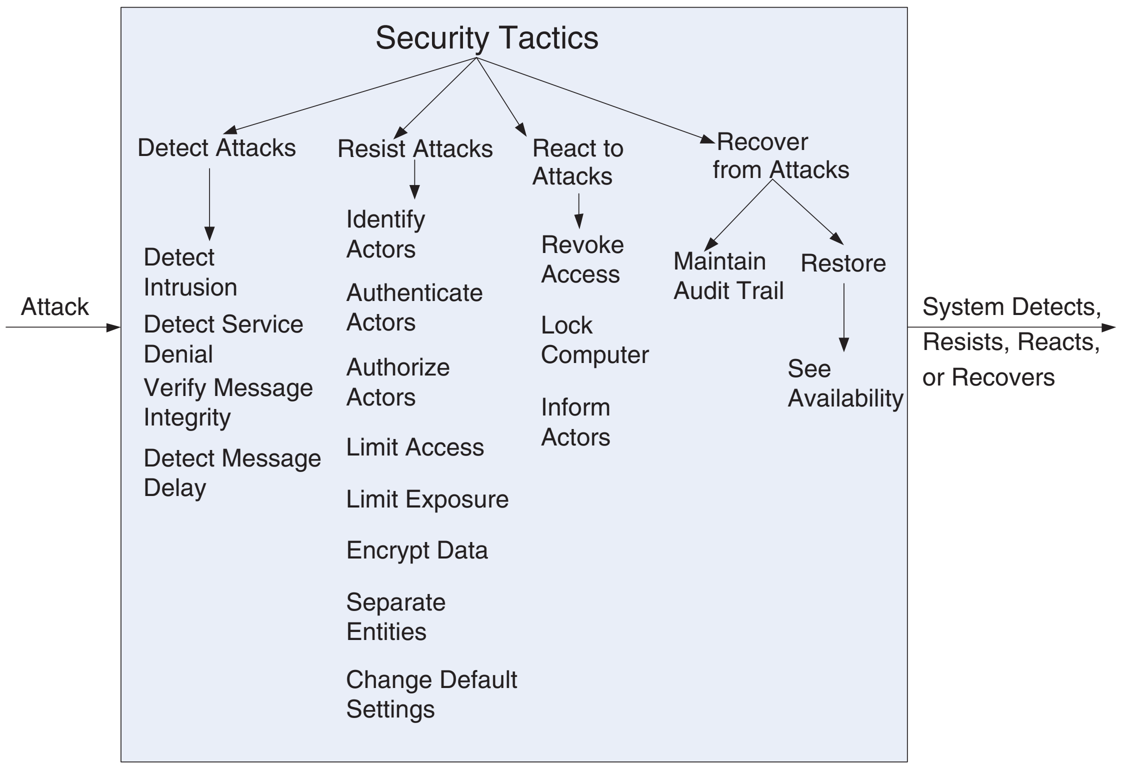


FIGURE 9.3 Security tactics

9.3 A Design Checklist for Security

Table 9.2 is a checklist to support the design and analysis process for security.

TABLE 9.2 Checklist to Support the Design and Analysis Process for Security

Category	Checklist
Allocation of Responsibilities	Determine which system responsibilities need to be secure. For each of these responsibilities, ensure that additional responsibilities have been allocated to do the following: ▪ Identify the actor ▪ Authenticate the actor ▪ Authorize actors ▪ Grant or deny access to data or services ▪ Record attempts to access or modify data or services ▪ Encrypt data ▪ Recognize reduced availability for resources or services and inform appropriate personnel and restrict access ▪ Recover from an attack ▪ Verify checksums and hash values

Category	Checklist
Coordination Model	Determine mechanisms required to communicate and coordinate with other systems or individuals. For these communications, ensure that mechanisms for authenticating and authorizing the actor or system, and encrypting data for transmission across the connection, are in place. Ensure also that mechanisms exist for monitoring and recognizing unexpectedly high demands for resources or services as well as mechanisms for restricting or terminating the connection.
Data Model	Determine the sensitivity of different data fields. For each data abstraction: ▪ Ensure that data of different sensitivity is separated. ▪ Ensure that data of different sensitivity has different access rights and that access rights are checked prior to access. ▪ Ensure that access to sensitive data is logged and that the log file is suitably protected. ▪ Ensure that data is suitably encrypted and that keys are separated from the encrypted data. ▪ Ensure that data can be restored if it is inappropriately modified.
Mapping among Architectural Elements	Determine how alternative mappings of architectural elements that are under consideration may change how an individual or system may read, write, or modify data; access system services or resources; or reduce availability to system services or resources. Determine how alternative mappings may affect the recording of access to data, services or resources and the recognition of unexpectedly high demands for resources. For each such mapping, ensure that there are responsibilities to do the following: ▪ Identify an actor ▪ Authenticate an actor ▪ Authorize actors ▪ Grant or deny access to data or services ▪ Record attempts to access or modify data or services ▪ Encrypt data ▪ Recognize reduced availability for resources or services, inform appropriate personnel, and restrict access ▪ Recover from an attack
Resource Management	Determine the system resources required to identify and monitor a system or an individual who is internal or external, authorized or not authorized, with access to specific resources or all resources. Determine the resources required to authenticate the actor, grant or deny access to data or resources, notify appropriate entities (people or systems), record attempts to access data or resources, encrypt data, recognize inexplicably high demand for resources, inform users or systems, and restrict access. For these resources consider whether an external entity can access a critical resource or exhaust a critical resource; how to monitor the resource; how to manage resource utilization; how to log resource utilization; and ensure that there are sufficient resources to perform the necessary security operations. Ensure that a contaminated element can be prevented from contaminating other elements. Ensure that shared resources are not used for passing sensitive data from an actor with access rights to that data to an actor without access rights to that data.

continues

TABLE 9.2 Checklist to Support the Design and Analysis Process for Security, *continued*

Category	Checklist
Binding Time	Determine cases where an instance of a late-bound component may be untrusted. For such cases ensure that late-bound components can be qualified; that is, if ownership certificates for late-bound components are required, there are appropriate mechanisms to manage and validate them; that access to late-bound data and services can be managed; that access by late-bound components to data and services can be blocked; that mechanisms to record the access, modification, and attempts to access data or services by late-bound components are in place; and that system data is encrypted where the keys are intentionally withheld for late-bound components
Choice of Technology	Determine what technologies are available to help user authentication, data access rights, resource protection, and data encryption. Ensure that your chosen technologies support the tactics relevant for your security needs.

9.4 Summary

Attacks against a system can be characterized as attacks against the confidentiality, integrity, or availability of a system or its data. Confidentiality means keeping data away from those who should not have access while granting access to those who should. Integrity means that there are no unauthorized modifications to or deletion of data, and availability means that the system is accessible to those who are entitled to use it.

The emphasis of distinguishing various classes of actors in the characterization leads to many of the tactics used to achieve security. Identifying, authenticating, and authorizing actors are tactics intended to determine which users or systems are entitled to what kind of access to a system.

An assumption is made that no security tactic is foolproof and that systems will be compromised. Hence, tactics exist to detect an attack, limit the spread of any attack, and to react and recover from an attack.

Recovering from an attack involves many of the same tactics as availability and, in general, involves returning the system to a consistent state prior to any attack.

9.5 For Further Reading

The architectural tactics that we have described in this chapter are only one aspect of making a system secure. Other aspects are these:

- *Coding. Secure Coding in C and C++* [Seacord 05] describes how to code securely. The Common Weakness Enumeration [CWE 12] is a list of the most common vulnerabilities discovered in systems.
- *Organizational processes.* Organizations must have processes that provide for responsibility for various aspects of security, including ensuring that systems are patched to put into place the latest protections. The National Institute of Standards and Technology (NIST) provides an enumeration of organizational processes [NIST 09]. [Cappelli 12] discusses insider threats.
- *Technical processes.* Microsoft has a life-cycle development process (The Secure Development Life Cycle) that includes modeling of threats. Four training classes are publicly available. www.microsoft.com/download/en/details.aspx?id=16420

NIST has several volumes that give definitions of security terms [NIST 04], categories of security controls [NIST 06], and an enumeration of security controls that an organization could employ [NIST 09]. A security control could be a tactic, but it could also be organizational, coding-related, or a technical process.

The attack surface of a system is the code that can be run by unauthorized users. A discussion of how to minimize the attack surface for a system can be found at [Howard 04].

Encryption and certificates of various types and strengths are commonly used to resist certain types of attacks. Encryption algorithms are particularly difficult to code correctly. A document produced by NIST [NIST 02] gives requirements for these algorithms.

Good books on engineering systems for security have been written by Ross Anderson [Anderson 08] and Bruce Schneier [Schneier 08].

Different domains have different specific sets of practices. The Payment Card Industry (PCI) has a set of standards intended for those involved in credit card processing (www.pcisecuritystandards.org). There is also a set of recommendations for securing various portions of the electric grid (www.smartgridipedia.org/index.php/ASAP-SG).

Data on the various sources of data breaches can be found in the Verizon 2012 Data Breach Investigations Report [Verizon 12].

John Viega has written several books about secure software development in various environments. See, for example, [Viega 01].

9.6 Discussion Questions

1. Write a set of concrete scenarios for security for an automatic teller machine. How would you modify your design for the automatic teller machine to satisfy these scenarios?
2. One of the most sophisticated attacks on record was carried out by a virus known as Stuxnet. Stuxnet first appeared in 2009 but became widely known in 2011 when it was revealed that it had apparently severely damaged or incapacitated the high-speed centrifuges involved in Iran's uranium enrichment program. Read about Stuxnet and see if you can devise a defense strategy against it based on the tactics in this chapter.
3. Some say that inserting security awareness into the software development life cycle is at least as important as designing software with security countermeasures. What are some examples of software development processes that can lead to more-secure systems?
4. Security and usability are often seen to be at odds with each other. Security often imposes procedures and processes that seem like needless overhead to the casual user. But some say that security and usability go (or should go) hand in hand and argue that making the system easy to use securely is the best way to promote security to the user. Discuss.
5. List some examples of critical resources for security that might become exhausted.
6. List an example of a mapping of architectural elements that has strong security implications. Hint: think of where data is stored.
7. Which of the tactics in our list will protect against an insider threat? Can you think of any that should be added?
8. In the United States, Facebook can account for more than 5 percent of all Internet traffic in a given week. How would you recognize a denial-of-service attack on Facebook.com?
9. The public disclosure of vulnerabilities in production systems is a matter of controversy. Discuss why this is so and the pros and cons of public disclosure of vulnerabilities.