Software Engineering Institute

Realizing and Refining Architectural Tactics: Availability

James Scott, Boeing Corporation
Rick Kazman, Software Engineering Institute

August 2009

TECHNICAL REPORT
CMU/SEI-2009-TR-006
ESC-TR-2009-006

Research, Technology, and System Solutions

http://www.sei.cmu.edu

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

Table of Contents

List of Figures

List of Tables

Acknowledgments

The authors wish to thank Len Bass, Robert Nord, and Joe Batman of the Carnegie Mellon® Software Engineering Institute for their many enlightening discussions regarding the relationships between tactics and quality attributes and for the categorization of quality attribute effects of tactics on design decisions.

Abstract

Architectural tactics are fundamental design decisions. They are the building blocks for both architectural design and analysis. A catalog of architectural tactics has now been in use for several years in academia and industry. This report illustrates the use of this catalog in industrial applications, describing how tactics can be used in both design and analysis. The report further shows how the needs of practice have caused the catalog of availability tactics to be updated, but demonstrates that the underlying structure of the tactics categorization has remained stable. Finally, a real-world example is provided of the application of the updated set of availability tactics, showing how applying tactics illuminates design decisions, as guided by associated heuristics and analytic models.

1 Introduction

The process of designing and analyzing software architectures is complex. Architectures are determined by requirements (principally quality attribute requirements), which are in turn determined by an organization's business goals and constraints. But moving from the domain of requirements to the domain of architecture has historically been an art more than a science. Architectural design is a minimally constrained search through a vast multi-dimensional space of possibilities. The end result is that architects are seldom confident that they have done the job optimally, or even satisfactorily.

Architectural patterns and styles have been proposed as a way to manage the unconstrained nature of the architectural design process and to reduce the enormous size and complexity of the search space [Garlan 1996, Buschmann 1996]. Such management has simplified the architectural design process somewhat, but it is still a challenge: styles and patterns have replaced one form of black magic—architectural design—with another form—choosing, tailoring, combining, and understanding patterns. Patterns are complex and their interactions with other patterns are not always clear. Furthermore, patterns are *always* underspecified, and so the designer still needs to add in considerable amounts of detail to reify these into an implementable design.

Patterns come with associated rationale, benefits, and liabilities (e.g., the Broker pattern is reported to have low fault tolerance, "restricted" efficiency, and to be difficult to test and debug). But such claims are contextual, depending on many environmental factors and detailed implementation decisions.

We have proposed a more fine-grained approach to architectural design, employing *tactics* [Bass 2003]. Tactics are the building blocks of architectures, and hence the building blocks of architectural patterns. We have defined sets of tactics that address six quality attributes: performance, usability, availability, modifiability, testability, and security. We have used these tactics over the past five years, as a foundation for designing and analyzing architectures. So, for example, tactics can ameliorate some of the deficiencies outlined above for the Broker pattern. The low fault tolerance of the "vanilla" Broker pattern could be ameliorated by using some form of active redundancy, for instance.[1]

Before we discuss availability tactics in detail let us first look at an example of *Modifiability* tactics to illustrate their impact [Bachmann 2007]. There are three classes of modifiability tactics:

1. those that *defer binding time decisions*, to control deployment time and cost
2. those that help to *localize changes*, reducing the number of modules directly affected by a change
3. those that *prevent ripple effects*, limiting the modifications to localized modules

[1] As will be described in Section 2.3, in one form of the *active redundancy* configuration, a group of processing nodes comprises both active and redundant nodes that receive and process identical inputs in parallel, maintaining a synchronous state and enabling instantaneous recovery and repair.

To make an architecture more modifiable, the designer needs to select and realize one or more tactics from this set.

Patterns package a number of tactics. Let us examine the most common architectural pattern—the Layered Pattern—to see how this works in practice. Layers group together similar sets of functionality and separate them from other functions that are expected to change independently. Through this separation, the modifiability of the system is expected to increase. For example, layering is often used to insulate a system from changes in the underlying platform (hardware and software), increasing maintainability while reducing integration and verification costs. This has been known at least as far back as Dijkstra's THE operating system [Dijkstra 1968]. To bring about this insulation, the architect creates one or more platform-specific layers to abstract the details of the underlying hardware and operating system. The rest of the system's functionality then accesses the underlying platform via these abstractions. To achieve this effect, the Layered Pattern employs two *Localize Change* tactics—*Semantic Coherence* and *Abstract Common Services*—to increase cohesion, and it employs three *Prevent Ripple Effects* tactics—*Use Encapsulation,Use an Intermediary*, and *Restrict Communication Paths*—to reduce coupling [Bachmann 2007].

What is the point of this more granular representation of design operations? A tactic is a design decision that is influential in the control of a *single* quality attribute response. As such it is simple to understand and analyze—its properties and effects are well understood. A pattern, on the other hand, is a prepackaged solution to a recurring problem that resolves *multiple* forces. Patterns are more complex and so it is much harder to understand the implications of changing the pattern. To understand a pattern, to tailor it, and to analyze it, you need to understand the tactics from which it is composed and the effects of each constituent tactic.

Returning to the Layered Pattern, if the modifiability of the system with respect to a specific responsibility needs to be increased, one tactic that could be employed is *Use an Intermediary*. Employing this tactic modifies the design: the independent functionality and the dependent functionality are now separated by a third component—the intermediary. Along with the tactic there is an analysis model (perhaps encapsulated in a "reasoning framework" [Bass 2005]) that allows the designer and the analyst to reason about the cost of a change both with and without the intermediary, and so make a reasoned decision. The tactic requires effort, and hence cost, to implement and maintain. In addition, after the implementation of this tactic, the strength of coupling between the dependent and independent functionality is reduced.

Every design decision has side effects. Once the *Use an Intermediary* tactic is in place, it will have an effect on runtime performance. Each of these attributes—cost, coupling, and performance impact—can be estimated by the architect and a reasoned decision can be made on whether to use the tactic.

In the remainder of this report, we will show how tactics are used in practice and how they inform both design and analysis. In particular we will show how availability tactics have been used and how they have been augmented over time to meet the needs of a changing world.

1.1 Using Tactics in Practice

Tactics can be used in both design and in analysis. They can be used in the design process to make decisions or, more commonly, to modify an architectural pattern. In this way, tactics aid in

enumerating and choosing among design decisions. Similarly, tactics can be used in analysis. Each tactic is easier to understand in isolation than a pattern and, as described by Bachmann, analysis models can be associated with tactics [Bachmann 2007]. For example, there is a formula for determining the increase in availability by adding a redundant hot spare. The architect, using this formula, can then reason about the costs and benefits of using this form of redundancy. Similarly the *Ping/Echo* tactic can be analyzed in terms of how long it takes to detect a fault (based on the period of the Ping and the number of missed Pings before a fault is determined to have occurred) and how the overhead of Ping messages degrades end-to-end latency in the system. To show how this reasoning is supported, a specific set of tactics—availability tactics—is now discussed in detail.

2 Tactics for Availability

Availability tactics are designed to enable a system to endure system faults such that a service being delivered by the system remains compliant with its specification [Bass 2003]. Inherent in this definition is the distinction between a system fault and a system failure. System faults are escalated to system failures once services are impacted to the point where they no longer comply with their specifications. 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.

System availability builds upon the concept of system reliability by adding the notion of recovery, which may be accomplished by fault masking, repair, or component replacement [Jalote 1994]. In practice, system requirements for availability are developed in accordance with steady-state availability (as opposed to instantaneous availability). Steady-state availability is the measurement of a system's uptime over a sufficiently long (90 days, one year, total mission, etc.) duration. The well-known expression used to derive steady-state availability is

$$\alpha = MTBF/(MTBF + MTTR)$$

Where MTBF refers to the *mean time between failures* (derived based on the expected value of the system's fault probability density function) and MTTR refers to the *mean time to repair* (which varies according to the repair strategy employed).

Table 1: System Availability Requirements

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

In practice, system designers develop a fault tree to characterize system faults according to their severity and service impact, and identify a suitable repair strategy for each branch of the tree. Table 1 provides an example of typical 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% ("5 nines") or greater. It should be noted that by definition, only unscheduled outages contribute to system downtime.

2.1 Updating the Tactics Catalog

A categorization of availability tactics provided by Bass is reproduced in Figure 1, below [Bass 2003]. As illustrated, availability tactics are categorized according to whether they address fault detection, recovery, or prevention.

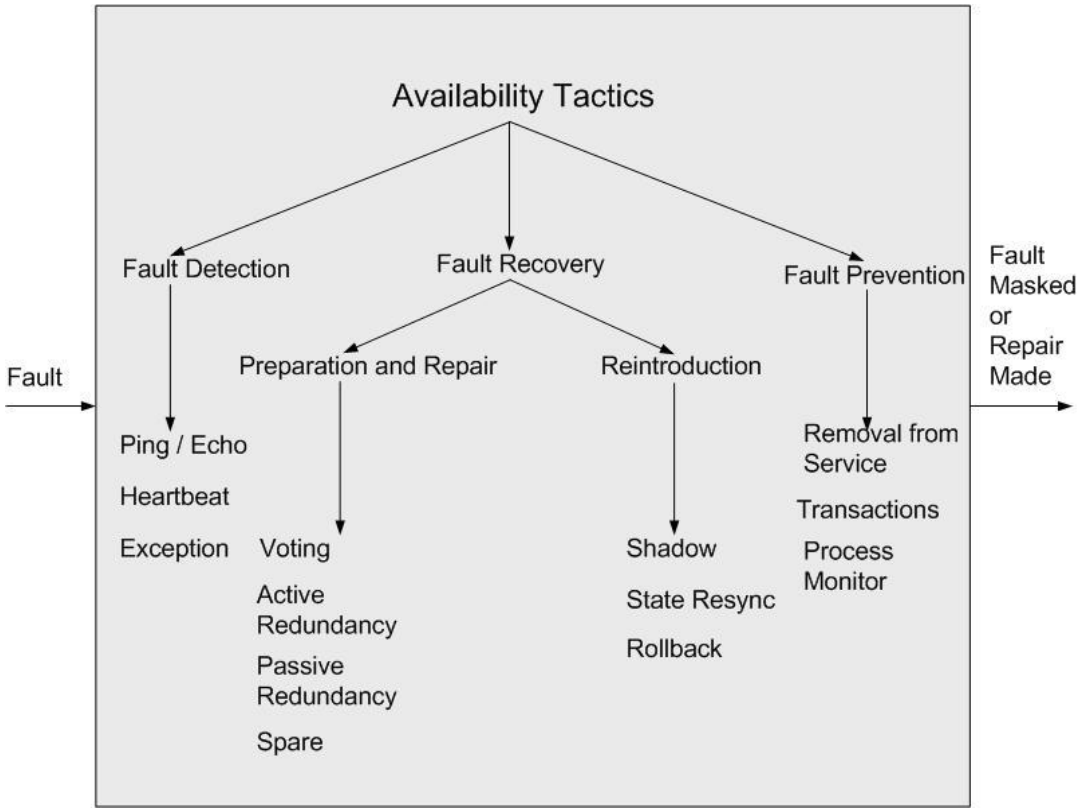

Figure 1: Original Availability Tactics

We will review the availability tactics described by Bass [Bass 2003] and then show a new version of this categorization that has been augmented and refined, based on several years of industrial experience using the categorization for the analysis and design of high-availability systems.

Figure 2 illustrates the refined view of the tactics outlined in Figure 1. In addition to refining the categorization, Figure 2 shows, below the tactics, some examples of specific implementation techniques for each.

2.2 Fault Detection Tactics

The tactics that were classified as being for fault detection were *Ping/Echo*, *Heartbeat*, and *Exception*. In addition to these tactics, reported by Bass [Bass 2003], we have classified *Voting* as a tactic whose primary purpose is fault detection.

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. Standard implementations of *Ping/Echo* are available for nodes interconnected via IP (*ICMP*

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

In addition we have generalized the notion of the *Heartbeat* tactic to *System Monitor*. In a high-availability system, a *System Monitor* tactic is used to monitor state of health, which includes the detection of hung or runaway processes; a heartbeat is one measure of health that a *System Monitor* could observe. 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, commonly referred to as "petting the watchdog."

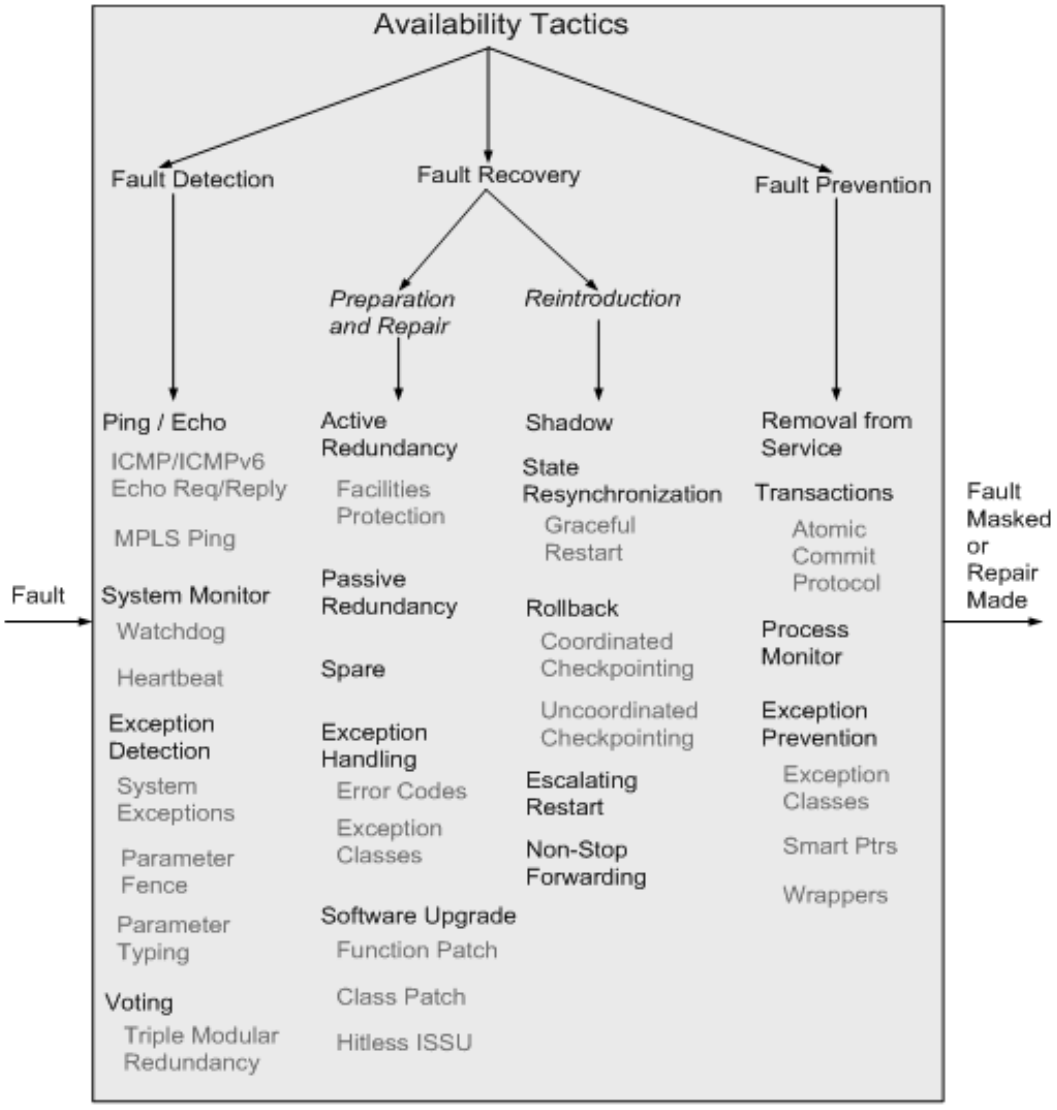

Figure 2: Refined Availability Tactics, with Examples

Table 2: Availability Tactics: Fault Detection

Tactic Control/click on tactic to go to its description in report	**Mechanism**	**Result**
Ping/Echo	asynchronous request/response message pair exchanging active nodes	determines reachability and round-trip delay through the associated network path
ICMP/ICMPv6 Echo Req/Reply		
MPLS Ping		
System Monitor		In high-availability system, monitors state of system health; detects hung or runaway processes
Watchdog	Hardware-based counter-timer periodically reset by software	Upon expiration, indicates to system monitor of a fault incurrence in the process
Heartbeat	Periodic message exchange between the system monitor and process	Indicates to system monitor when fault is incurred in process
Exception Detection		Detects system conditions that alter the normal flow of execution
System Exceptions	System raises an exception when it detects a fault	Detects faults such as divide by zero, bus and address faults, illegal program instructions
Parameter Fence	Incorporates an a priori data pattern (such as 0xdeadbeef) placed immediately after any variable-length parameters of an object	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	Uses strong typing to build and parse messages

Tactic Control/click on tactic to go to its description in report	Mechanism	Result
Voting		
Triple Modular Redundancy	Three identical processing units, each receiving identical inputs, whose output is forwarded to voting logic	Detects any inconsistency among the three output states, which is treated as a system fault

Expiration of the watchdog counter/timer provides an indication to the *System Monitor* that the process being monitored has incurred a fault. When the underlying fault detection mechanism employs a periodic message exchange between the *System Monitor* and the process being monitored, this is referred to as a *Heartbeat*. For larger systems where scalability is a concern, transport and processing overhead efficiency can be increased by piggybacking *Heartbeat* messages on to other control messaging being exchanged between the process being monitored and the distributed system controller. In this case, there is an added dependency between the Messaging System and the *System Monitor*. Based on the above discussion, we revise the *Fault Detection* tactic category described by Bass [Bass 2003] to include a *System Monitor* tactic that is further refined to include a *Watchdog* and a *Heartbeat* tactic.

The *Voting* fault detection tactic is based on the fundamental contributions to automata theory by Von Neumann, who demonstrated how systems having a prescribed reliability could be built from unreliable components [Von Neumann 1956]. The common realization of this tactic is referred to as *Triple Modular Redundancy* (TMR), which employs three identical processing units, each of which receives identical inputs, and forwards their output to voting logic, used to detect any inconsistency among the three output states. Any such inconsistency is treated as a system fault. TMR depends critically on the voting logic, which can be realized either as a singleton where the probability of error is sufficiently low (the voting logic is a simple Boolean AND/OR combination) or as a redundant triple [Lyons 1962]. To demonstrate the improvement in system reliability (specifically, MTBF) of a TMR-based design, consider a system where the probability of error of a single bit is defined as ε_e. Applying TMR to that single bit will reduce the probability of error from ε_e to

$$\varepsilon_{TMR} = 3\varepsilon_e^2(1-\varepsilon_e) + \varepsilon_e^3$$

For example, if a single component has an error rate of .001, a TMR version of this component will have an error rate of 0.000002998, or about three orders of magnitude better. TMR is commonly realized at the hardware gate and chip level, but can also be employed in software at the thread or process level for scenarios where the outputs of the multiple threads or processes can be synchronized by the voting logic.

The final *Fault Detection* tactic identified by Bass is the *Exception* tactic [Bass 2003]. The *Exception* tactic can be further refined into *Exception Detection*, *Exception Handling*, and *Exception Prevention* tactics. *Exception Detection* refers to the detection of a system condition that alters

the normal flow of execution. For distributed real-time embedded systems, the *Exception Detection* tactic can be further refined to include *System Exceptions*, *Parameter Fence*, and *Parameter Typing* tactics. *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 [Utas 2005]. *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 will result in higher availability than implementations that simply treat messages as byte buckets [Utas 2005].

2.3 Fault Recovery Tactics

Fault Recovery tactics are refined into *Preparation and Repair* tactics and *Reintroduction* tactics. *Preparation and Repair* tactics include *Active Redundancy*, *Passive Redundancy*, *Spare*, *Exception Handling*, and *Software Upgrade*.[2] *Reintroduction* tactics include *Shadow*, *Rollback*, *Escalating Restart*, and *Non-Stop Forwarding*.

High-availability distributed real-time embedded systems commonly employ a strategy of equipment protection, where spatially redundant line cards (circuit packs) are employed in a hot, warm, or cold sparing configuration. These three configurations are referred to by Bass as *active redundancy* (hot sparing), *passive redundancy* (warm sparing), and simply *sparing* (cold sparing) [Bass 2003]. In our updated catalog of availability tactics, we refer to cold sparing as the *Spare* tactic. Before describing each of these three configurations, we first define a protection group as being a group of processing nodes where one or more nodes are "active" with the remaining nodes in the protection group serving as redundant spares.

Active Redundancy refers to a configuration where all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). Because the redundant spare possesses an identical state to the active processor, recovery and repair can occur in time measured in 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. Standards-based realizations of *Active Redundancy* exist for protecting network links (i.e., facilities) at both the physical layer [Bellcore 1998, 1999, Telcordia 2000] and network/link layer [IETF 2005].

Passive Redundancy refers to a configuration where only the active members of the protection group process input traffic, with the redundant spare(s) receiving 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 check-

[2] Many tactics span multiple categories. For example, *Voting* can be considered a *Fault Detection* tactic, (detecting a dissenting vote), or a *Fault Preparation and Repair* tactic (correlating the fault). Voting also aids in *Fault Prevention* (by identifying a processor to be reset or repaired). However, in the taxonomy we have chosen to include it within the *Fault Detection* category.

pointing 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 plex *Active Redundancy* tactic and the less available but significantly less complex *Spare* tactic.

Cold sparing, or simply *sparing* in the nomenclature of Bass, 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 [Bass 2003]. 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.

In practice, the system architect will determine whether to use *Active Redundancy*, *Passive Redundancy*, or *Spare* based on the system availability requirements allocated. Figure 3 illustrates the data flow for each of these three tactics, embedded in the context of patterns.

Recall from Section 2.2 that the *Exception* tactic can be refined into *Exception Detection, Exception Handling*, and *Exception Prevention* tactics, with *Exception Detection* being discussed in that section. 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 [Powell-Douglas 1999].

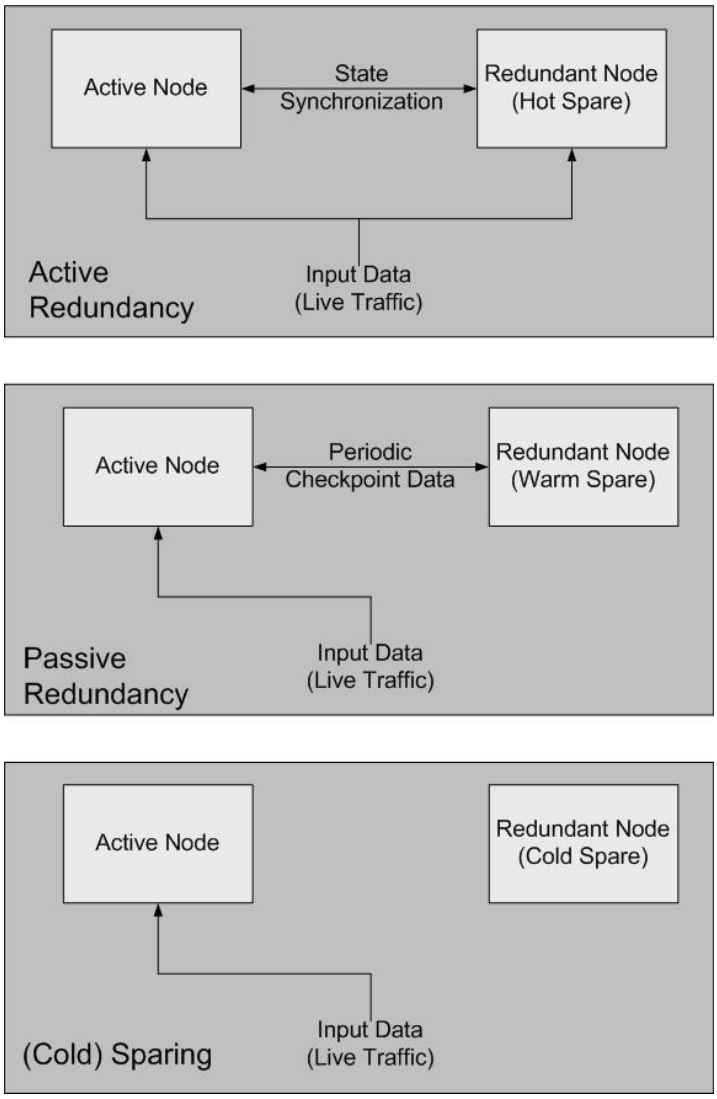

Figure 3: System Redundancy Tactics Embedded in Patterns

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 [Scott 2008]. This tactic is refined by *Function Patch*, *Class Patch*, and *Hitless In-Service Software Upgrade* (ISSU) tactics. The *Function Patch* tactic is used in a procedural programming environment 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 backdoor mechanism that enables the runtime addition of member data and functions. *Hitless In-Service Software Upgrade* is a tactic that 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* tactics are used to deliver bug fixes while the *Hitless In-Service Software Upgrade* tactic is used to deliver new features and capabilities.

Table 3: Availability Tactics: Fault Recovery

Tactic Control/click on tactic to go to its description in report.	Mechanism	Result
Preparation and Repair		
Active Redundancy	Configuration wherein all of the nodes (active or redundant spare) in a protection group receive and process identical inputs in parallel	Redundant spare possesses an identical state to the active processor, so recovery and repair can occur in milliseconds
Passive Redundancy	Configuration wherein only the active members of the protection group process input traffic, with the redundant spare(s) receiving periodic state updates	Achieves a balance between the more highly available but more complex *Active Redundancy* tactic and the less available but significantly less complex *Spare* tactic
Spare	Configuration wherein the redundant spares of a protection group remain out of service until a fail-over occurs	Initiates power-on-reset procedure on the redundant spare prior to its being placed in service
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	
Exception Classes	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	Allows system to transparently recover from system exceptions
Software Upgrade		Achieves in-service upgrades to executable code images in a non-service affecting manner
Function Patch	An incremental linker/loader in a procedural programming environment	Stores an updated software function into a pre-allocated segment of target memory

Tactic Control/click on tactic to go to its description in report.	**Mechanism**	**Result**
Class Patch	Applicable for targets executing object-oriented code, where the class definitions include a backdoor mechanism	Enables the runtime addition of member data and functions
Hitless In-Service Software Upgrade (ISSU)	Leverages the *Active Redundancy* or *Passive Redundancy* tactics	Achieves non-service-affecting upgrades to software and associated schema
Reintroduction		Failed component is reintroduced after correction
Shadow	Operates a previously failed or in-service upgraded component in a "shadow mode" for a pre-defined duration of time	Reverts the component back to an active role
State Resynchronization	When realized as a refinement to the Active Redundancy tactic, occurs organically as active and standby components each receive and process identical inputs in parallel When realized as a refinement to the Passive Redundancy tactic, is based solely on periodic state information transmitted from the active component(s) to the standby component and involves the *Rollback* tactic	Allows a system's control element to dynamically recover its control plane state from its network peers Periodically compares state of the active and standby components to ensure synchronization
Graceful Restart	Includes inter-networking devices such as cross-connects, switches, and packet routers	Allows a system's control element to dynamically recover its control plane state from its network peers
Rollback	Checkpoint based	Allows the system state to be reverted to the most recent consistent set of checkpoints
Coordinated Checkpointing	Allows processes to resolve dependencies and restart at a coordinated checkpoint	More complex mechanism that is always consistent

Tactic Control/click on tactic to go to its description in report.	Mechanism	Result
Uncoordinated Checkpointing	Allows processes to take checkpoints when most convenient	Simple mechanism but may not always result in a consistent state
Escalating Restart	Varies the granularity of the component(s) restarted and minimizes the level of service affectation	At final level, completely reloads and reinitializes the executable image and associated data segment
Non-Stop Forwarding	Includes inter-networking devices such as cross-connects, switches, and packet routers	Maintains proper functioning of the user data plane (bearer channel services)

Other *Preparation and Repair* tactics exist that are primarily realized by a hardware design and, hence, beyond the scope of this work. 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 1980]. Conversely, FEC coding is typically employed to recovery from physical layer errors occurring on external network links [Morelos-Zaragoza 2006]. *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 2002].

Some *Fault Recovery* tactics rely on component reintroduction, where a failed component is reintroduced after it has been corrected. *Reintroduction* tactics include *Shadow*, *State Resynchronization/Graceful Restart*, *Rollback*, *Escalating Restart*, and *Non-Stop Forwarding*.

The *Shadow* tactic refers to operating a previously failed or in-service upgraded component in a "shadow mode" for a pre-defined duration of time prior to reverting the component back to an active role. In this context, the *Shadow* tactic is a *Reintroduction* version of the *Hitless In-Service Software Upgrade* tactic previously discussed as a *Preparation and Repair* tactic.

Similarly, *State Resynchronization* is a reintroduction refinement to the *Active Redundancy* and *Passive Redundancy* preparation and repair tactics. When realized as a refinement to the *Active Redundancy* tactic, the *State Resynchronization* occurs organically, as 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 [Morelos-Zaragoza 2006] or, for systems providing safety-critical services, a message digest calculation (a one-way hash function) [Schneier 2001]. Conversely, when realized as a refinement to the *Passive Redundancy* (warm sparing) tactic, *State Resynchronization* is based solely on periodic state information transmitted from the active component(s) to the standby component(s). This operation involves an additional reintroduction tactic referred to as *Rollback*. *Rollback* is a checkpoint-based recovery mechanism

that allows the system state to be reverted to the most recent consistent set of checkpoints. The set of checkpoints is referred to as a "recovery line," which may be generated using graph-theoretic techniques described by Elnozahy [Elnozahy 2002]. Checkpoint-based rollback recovery may employ *Uncoordinated Checkpointing*, where processes are allowed to take checkpoints when most convenient, or *Coordinated Checkpointing*, where processes resolve dependencies and restart at a coordinated checkpoint [Scott 2008]. A specialization of *State Resynchronization*, used in tandem with the *Non-Stop Forwarding* tactic is *Graceful Restart*, which allows a system's control element to dynamically recover its control plane state from its network peers. Standard realizations of *Graceful Restart* have emerged for a variety of commonly deployed routing and signaling protocols such as BGP [IETF 2007b, 2007c], OSPF [IETF 2007a][IETF 2008], LDP [IETF 2003b] and RSVP-TE [IETF 2003a, 2003c].

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 affectation [Utas 2005]. For example, consider a system that supports four levels of restart, as follows. The lowest level of restart (call it Level 0), and hence least impacting on services, employs *Passive Redundancy* (warm restart), where all child threads of the component in which the fault was detected 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.

Another *Reintroduction* tactic used to enable graceful degradation of high-availability systems is *Non-Stop Forwarding* (NSF). This concept is borrowed from commercial best current practice in designing inter-networking devices, such as cross-connects, switches, and packet routers. *Non-Stop Forwarding* refers to the ability of the device to maintain proper functioning of the user data plane (bearer channel services), even when the device's control and/or management planes are out of service. Support for *Non-Stop Forwarding* implies a strict separation of control/management and data plane functionality in the system design, as described through the Internet Engineering Task Force [IETF 2004]. In heritage digital cross-connect (DXC) and ATM switch design, this tactic is referred to as the "Headless Mode" of operation. The term *Non-Stop Forwarding* has emerged as the standard nomenclature used when the *Headless Mode* tactic is applied to packet router designs targeting high-availability services.

2.4 Fault Prevention Tactics

Fault Prevention tactics include *Removal from Service*, *Transactions*, *Process Monitor*, and *Exception Prevention*. In the context of fault prevention, the *Removal from Service* tactic refers to placing a system component in an out-of-service state for the purpose of mitigating potential system failures. One example involves taking a component of a system out of service and resetting the component in order to scrub latent faults (such as memory leaks, fragmentation, or soft errors

in an unprotected cache) before the accumulation of faults become service affecting (resulting in system failure).

Systems targeting high-availability services leverage transactional semantics to ensure that asynchronous messages exchanged between distributed components are atomic, consistent, isolated, and durable [Gray 1993]. These four properties are referred to as the "ACID properties" and are generally a requirement for high-availability systems, particularly those that provide either mission-critical or safety-critical services. The *Transactions* tactic is typically realized using an "atomic commit protocol," the most common of which is the "two-phase commit" (a.k.a. 2PC) protocol, originally described by Gray [Gray 1993]. Figure 4 illustrates the successful case of a distributed two-phase commit transaction. For the case where a distributed two-phase commit transaction fails, the transaction coordinator will employ the *Rollback* tactic among all distributed components involved in the failed transaction, in order to ensure a consistent and durable system state.

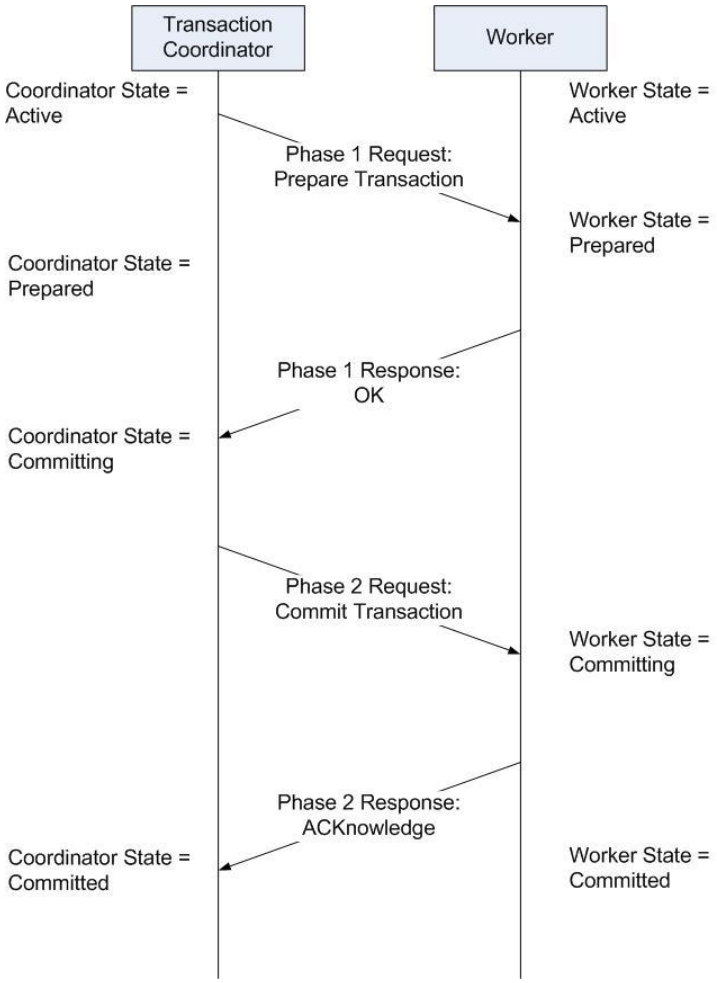

Figure 4: Transactions Tactic: Two-Phase Commit

In the context of fault prevention, the *Process Monitor* tactic is employed to monitor the state of health (SOH) of a system process in order to ensure that the system is operating within its nominal operating parameters. This tactic has a symbiotic relation to the *System Monitor* previously described, and in practice, the *Process Monitor* may be a lower level function of a hierarchical system monitoring function.

Recall from the previous discussion that the *Exception* tactic has been refined into *Exception Detection*, *Exception Handling*, and *Exception Prevention* tactics, with *Exception Detection* described in Section 2.2 and the *Exception Handling* tactic described in Section 2.3. The *Exception Prevention* 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 recovery from system exceptions, was discussed in Section 2.3 [Powell-Douglas 1999]. Other examples of tactical refinements used to realize *Exception Prevention* include abstract data types such as *Smart Pointers* and the use of *Wrappers* [Gamma 1995] to prevent faults such as dangling pointers and semaphore access violations from occurring.

Table 4: Availability Tactics: Fault Prevention

Tactic Control/click on tactic to go to its description in report	**Mechanism**	**Result**
Removal from Service	Places a system component in an out-of-service state	Allows for mitigating potential system failures before their accumulation affects service
Transactions		Ensures a consistent and durable system state
Atomic Commit Protocol	Most commonly the two-phase commit	
Process Monitor		Monitors the state of health of a system process; ensures that the system is operating within its nominal operating parameters
Exception Prevention		Prevents system exceptions from occurring
Exception Classes	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	Allows system to transparently recover from system exceptions
Smart Pointers/ Wrappers	Abstract data types that control all access through pointers	Prevent faults such as dangling pointers and semaphore access violations

3 An Example

We will now present an example from the internetworking domain, where a network node is responsible for providing high-availability services, such as cross-connecting voice circuits, switching cells, frames, or packets, or routing and forwarding IP packets.

The system architect, knowing that high availability is crucial, has determined that some form of redundancy will be employed. The architect's next step is to determine and apply the appropriate redundancy tactics to ensure compliance with the system's precise availability requirements. Architectural insight can and should be developed through model-based analysis wherever possible. Consider the case of an architecture being developed for a high-availability system having a 99.999% (a.k.a. "5 nines") availability requirement. We will show how we employ a Markov model to determine the appropriate redundancy tactic to employ in the system architecture.

3.1 The Availability Model

The availability model, from Gokhale, employed and illustrated in Figure 5, takes into account the Mean Time Between Failure and Mean Time To Recover for both hardware and software components of the system [Gokhale 2005]. In addition, failures are characterized according to high and low severity levels. Each of the hardware and software components can be in one of three states: *In-Service, Degraded Service,* or *Out-of-Service*. *In-Service* implies nominal operation with no faults having been detected and with service levels consistent with the components' specifications. *Degraded Service* implies that a severity 2 (low priority) fault has been detected and correlated and is in the process of being mitigated. Mitigation strategies for severity 2 faults may range from resetting ASIC personalities or software processes to executing a managed fail-over to a redundant processor. *Out-of-Service* implies that a severity 1 (high priority) fault has been detected and correlated and is in the process of being mitigated. Mitigation of severity 1 faults typically involves a managed fail-over to a redundant processor. The combination of three states for hardware and software components results in a state space of nine states in the Markov model. The simultaneous severity 1 failure state (i.e., severity 1 faults in both hardware and software) is not considered to be a valid state, as the fault detection and correlation logic would assign the (root cause) fault to either software or hardware, but not both.

The model assumes that MTBF values for all four types of failures are exponentially distributed. MTBF values for severity 1 and 2 hardware and software failure states are denoted in the model using the $\{\lambda_{\text{HW-SEV1}}, \lambda_{\text{HW-SEV2}}, \lambda_{\text{SW-SEV1}}, \lambda_{\text{SW-SEV2}}\}$ nomenclature. Similarly, the model assumes that MTTR values for recovering from the four types of failures are exponentially distributed. MTTR values for recovering from severity 1 and 2 hardware and software failure states are denoted by the $\{\mu_{\text{HW-SEV1}}, \mu_{\text{HW-SEV1}}, \mu_{\text{SW-SEV1}}, \mu_{\text{SW-SEV1}}\}$ nomenclature. The steady-state system availability (A) is determined by adding the probabilities of the system operating in a nominal In-Service state and the various combinations of In-Service and Degraded Service states.

$$A = P(HW_{IS} : SW_{IS}) + P(HW_{IS} : SW_{SEV2}) + P(HW_{SEV2} : SW_{IS}) + P(HW_{SEV2} : SW_{SEV2})$$

In order to solve the preceding equation, a system of balance equations is derived from the availability model. The derivation of the balance equations and resulting solution for this model are found in Gokhale [Gokhale 2005].

Table 5 provides an analysis of system availability for three separate *Redundancy* tactics: *Active* (hot sparing), *Passive* (warm sparing), and *Spare* (cold sparing). The analysis assumes that software failures are five times more likely to occur than hardware failures and that severity 2 failures are also five times more likely to occur than severity 1 failures. The analysis assumes that the MTTR for severity 2 failures is 30 seconds for each scenario (active, passive, and spare) and that the MTTR for severity 1 failures is one second for *Active Redundancy*, five seconds for *Passive Redundancy*, and 15 minutes for the (cold sparing) *Spare* tactic. Wherever possible such assumptions can and should be validated by measurements of prototypes or existing systems.

From this analysis, the system architect is able to determine that an availability requirement of 99.999% can be met using the *Passive Redundancy* (warm sparing) tactic, albeit with no additional margin. Note also that this analysis also indicates the difficulty in achieving a more stringent 99.9999% (a.k.a. "6 nines") of availability, even when the more complex and more costly *Active Redundancy* tactic is used.

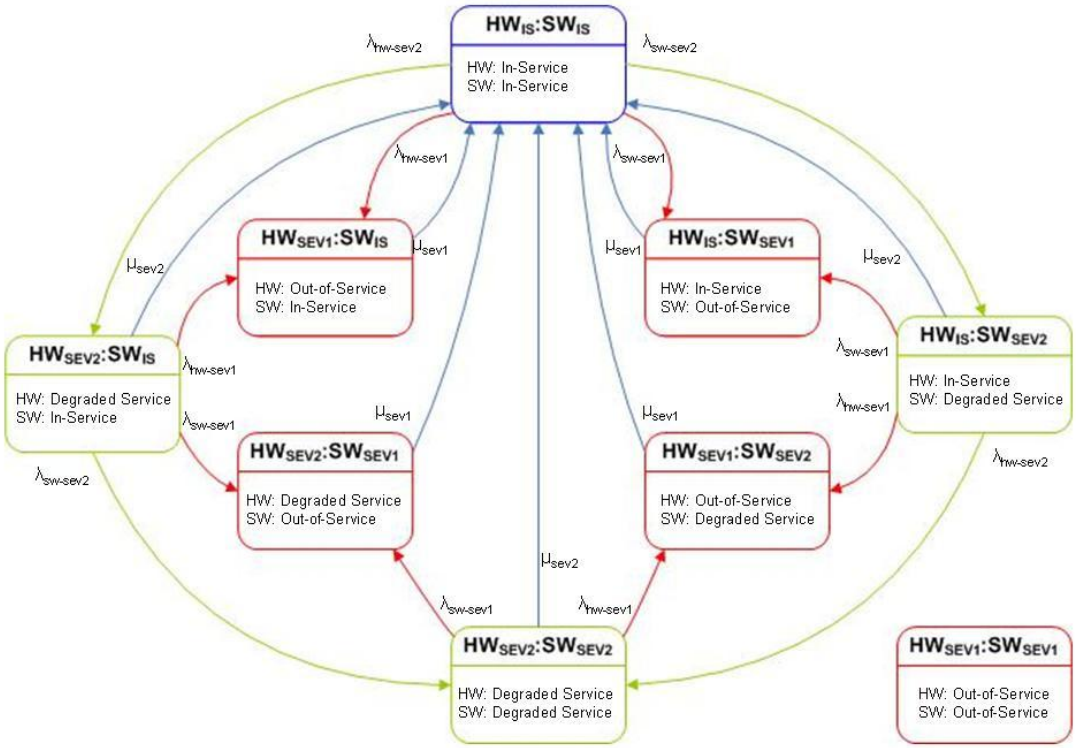

Figure 5: Markov Model of System Availability

Table 5: System Availability of Redundancy Tactic Used

Function	Failure Severity	MTBF (Hours)	MTTR_active (Sec)	MTTR_passive (Sec)	MTTR_sparing (Sec)
Hardware	1	250,000	1	5	900
	2	50,000	30	30	30
Software	1	50,000	1	5	900
	2	10,000	30	30	30
Availability			0.999998	0.999990	0.9982

3.2 The Resulting Redundancy Tactic

In this example we achieved equipment protection via the *Passive Redundancy* (warm sparing) tactic, where the active processor(s) in a protection group receives input traffic and transmits checkpointed state periodically to the associated redundant spare(s). In practice, this tactic achieves a reasonable balance between design complexity and performance for MTTR. Systems designed for high availability (99.999% uptime) commonly employ *Passive Redundancy*. Conversely, systems with more stringent availability requirements will necessarily employ the more complex *Active Redundancy* (hot sparing) tactic, to further reduce MTTR.

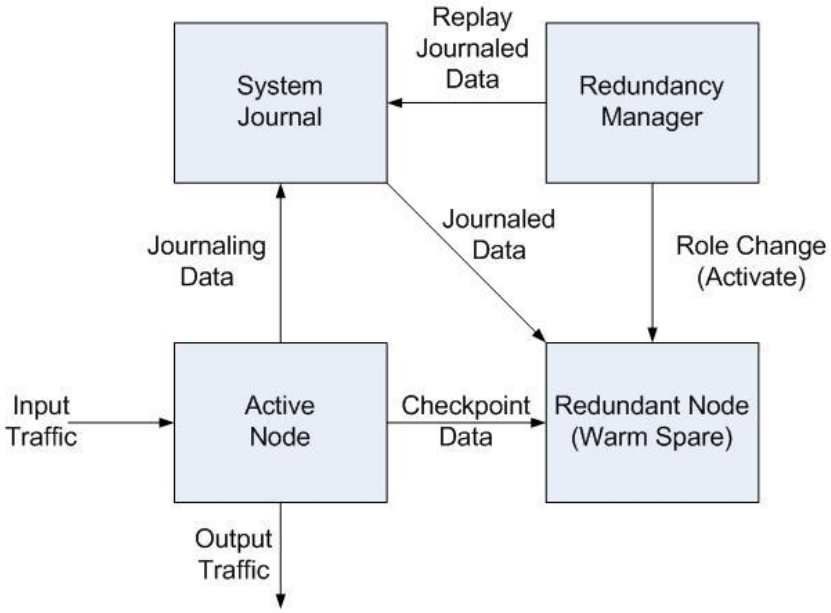

Figure 6: Passive Redundancy Tactics

A realization of the *Passive Redundancy* tactic, embedded in a pattern, is illustrated in Figure 6, where each of the boxes is a system node. The pattern employs four functions: the Active Node, the Redundant (passive/warm spare) Node, a Redundancy Manager, and a System Journal (a journaling mechanism). The Active and Redundant Nodes are identical copies of a system processor (or collection of processors). The Active Node periodically transmits checkpointed state data to the Redundant Node, to maintain a loosely synchronized state. The System Journal is employed to ensure that transient state not included in the checkpoint data set is not lost as a result of a role change (where the Redundant Node assumes the Active role). The Redundancy Manager is used to manage the assignment of (Active, Spare) roles to the processors in the protection group and detect the "liveness" of the currently designated Active Node, to identify the recovery line (i.e., the most recent consistent set of checkpointed data), and to manage the transmission of journaled data from the System Journal to a Redundant Node undergoing a role change (when assuming the

role of an Active Node). This pattern may additionally employ a *Reintroduction* tactic such as *Coordinated Checkpointing*, where the processes involved autonomously resolve checkpoint dependencies and restart at a coordinated checkpoint, or may use *Uncoordinated Checkpointing*, where each process is allowed to save a checkpoint when it is most convenient. As described in Elnozahy, *Uncoordinated Checkpointing* achieves a lower latency recovery for the nominal case but can be subjected to resolving a cascade of checkpoint dependency issues for the off-nominal case (referred to as the "Domino Effect") [Elnozahy 2002]. *Coordinated Checkpointing* eliminates the Domino Effect at a cost of slightly higher latency for each recovery (due to the processing penalty associated with central management of the distributed checkpoint data). An excellent survey of best practices for checkpoint rollback-recovery protocols is provided by Elnozahy [Elnozahy 2002].

For cases where either a singleton spare is used to protect multiple active nodes (1:N, pronounced "one for N") or multiple spares are used to protect multiple active nodes (M:N), the checkpoint and journal data can be saved to persistent (or redundancy-protected) memory, with the Redundancy Manager directing the Redundant Node as to which set of checkpoint and journaled data to retrieve upon a role change.

In addition to simplified design complexity (when compared to the stricter state synchronization requirements imposed by the *Active Redundancy* tactic), we find that the time overhead for the nominal case of *Passive Redundancy* is low, as it is a function solely of the time required to export the journaled (transient, uncheckpointed) data from the Active Node and to periodically calculate a new recovery line, save the data as a checkpoint, and transmit the checkpointed state to the Redundant Node. Similarly, the space overhead imposed by this tactic is low, bounded by the checkpointed state residing on the Active and Redundant Nodes and the transient journaled data maintained on the Active Node and on the node hosting the System Journal. And finally, the communication overhead imposed by this tactic for the nominal case is also low, bounded by the control messaging used to register with the System Journal and for the Redundancy Manager to (re)assign roles, and the replication of checkpoint and journaled data between Active Node and the Redundant Node and System Journal, respectively [Saridakis 2002].

3.3 Tactics Guide Architectural Decisions

The system architect needs to determine the appropriate availability tactic(s) to employ based on a consideration of the MTTR requirements for the various network services enabled by the device, as well as the side effects of the chosen tactics. For example, if a given processor in the system hosts a service with a requirement that service outages be repaired on sub-second timescales, then *Active Redundancy* (hot sparing) is the suitable availability tactic. Conversely, if that processor hosts a service that requires that outages be repaired in timescales measured in seconds, then *Passive Redundancy* (warm sparing) may be employed. And, if the system requirements allow for service outages to be repaired in timescales measured in minutes, then *Sparing* (cold sparing) is a suitable availability tactic to employ. Note also that some systems have both stringent availability and reliability requirements, in which case it may be necessary to employ either *Active Redundancy* or *Passive Redundancy* (to address the MTTR component of the availability requirement) in tandem with *Sparing* (to address system reliability requirements relating to Mean Mission Duration). Each of these tactics has side effects: *Active Redundancy* consumes more runtime resources (processing and communication) than *Passive Redundancy*, to keep the redundant components

synchronized. And both *Active* and *Passive Redundancy* increase the cost of the system more than *Sparing*. In each case the availability requirement, along with performance requirements and constraints on costs, guides the architect to a choice of tactics, even though the analysis is initially at a crude, often heuristic, level.

In determining the appropriate availability tactic(s) to employ, the system architect must consider the system availability requirement(s) and any associated availability sub-allocations for the constituent system components. Recall from the previous discussion that availability extends the concept of system reliability (defined by the MTBF) to include the notion of Mean Time to Recovery (MTTR). It is the system MTTR (or associated MTTR values sub-allocated to the system's constituent components) that determine the set of appropriate availability tactics to consider.

For the high-availability example provided, various architectural alternatives could be considered. For example, rather than employ the *Passive Redundancy* tactic, the architect could specify the more stringent *Active Redundancy* tactic, which would provide a greater capability with regard to system availability but at the risk of over-engineering the system (increasing cost and complexity) for the given set of requirements. Once *Passive Redundancy* had been selected as the suitable tactic, the size of the protection group had to be considered. For example, would the system employ a 1:1 form of equipment protection, where a single warm spare is employed for each active processor, or will the system employ a 1:N (one redundant spare used for N active processors) form, or even an M:N (M redundant spares used to protect N active processors) form of equipment protection?

Next, once the sparing model has been selected, the design space includes multiple options for realizing the *Reintroduction* tactic. For example, transactional semantics can be employed to ensure that the ACID properties are supported by the system's distributed database (which does not imply use of a relational database). Alternately, a checkpointing strategy could be used, leveraging either *Coordinated Checkpointing* or *Uncoordinated Checkpointing* along with one of the rollback recovery protocols described by Elnozahy [Elnozahy 2002]. For networking devices that host link-state protocols, which support the notion of peering with its neighboring nodes, *Reintroduction* tactics such as *Graceful Restart* and *Non-Stop Forwarding* could be employed in order to reacquire control plane state (from its peer) while maintaining fault-free user data services.

Verifying that the collection of tactics employed results in a system that complies with its availability requirements involves several levels of analysis. In practice, the system availability requirement is decomposed into separate sub-allocations for hardware- and software-induced failures. For hardware, a common approach for estimating system reliability (the MTBF component of availability) is to employ a Markov Chain to model the interconnection of the various hardware components, where the failure rate function for a given component follows a Weibull distribution. Modeling the system reliability for software-induced failures differs from the above approach due to the failure rate function for the constituent component's being more accurately represented by a Poisson distribution, due to the periodic introduction of new software releases. Popular analytical models used to estimate software reliability are based on Markov Chains, Non-Homogeneous Poisson Processes (NHPP), and Bayesian formulations [Musa 1987, Xie 1991].

The contribution of both hardware- and software-induced failures to system availability can be computed as the ratio of uptime to the sum of uptime and downtime, as the time interval over

which the measurement is made approaches infinity (as described in Section 3). Note that the downtime is the product of the failure intensity and the MTTR. To reiterate, only service-affecting failures contribute to the system availability.

4 Implications

We can draw a number of implications from this example. The most important is that while architectural design, even for a fragment of a system, is a complex search through a potentially unbounded space of possibilities, the use of tactics guides and constrains this search and makes it far more tractable. Each tactic can be associated with an analytic model (e.g., see the discussion in Section 3.1). So tactics work on two levels: similar to architectural patterns they guide the architect towards a particular solution—a choice of a tactic—but unlike architectural patterns they support a deeper (more precise) form of analysis based on analytic models. As shown in our example, these analyses may range from guidelines and heuristics to precise mathematical models, as dictated by the level of risk surrounding the architecture's realization of a quality attribute.

Tactics are the building blocks of patterns. A pattern, such as the one shown in Figure 6, is a composition of multiple tactics: *System Monitor*, *Passive Redundancy*, *State Resynchronization*, *Rollback*, and so on. Each of these may in turn be decomposed into even lower level tactics (*Heartbeat*, *Coordinated Checkpoint*, *Graceful Restart*, *Non-Stop Forwarding*, etc.).

5 Conclusions

This report has presented an update to the catalog of architectural tactics for availability. We have a similar update to the catalog for performance that space prevents us from presenting here. These updates have been motivated by practice—by observing, and categorizing, the sets of tactics in actual use. The structure of neither the availability nor the performance tactic catalog has changed dramatically since they were introduced seven years ago; this shows that the notion of tactics is robust—they are fundamental elements of design. Only the realizations of the tactics have changed, and this is to be expected as technologies mature.

This report has also shown how the catalog of tactics can be, and is, used in practice, to guide in making fundamental architectural design decisions that have implications in multiple dimensions. For each tactics-based design decision there are heuristics associated with the decision, as well as associated analytic models.

From this presentation we can see that tactics are useful in both design and analysis. They are useful because they restrict the design and analysis vocabulary, reduce the size of the search space, and directly suggest analytic models.

References

URLs are valid as of the publication date of this document.

[Bachmann 2007]
Bachmann, F.; Bass L.; & Nord, R. *Modifiability Tactics* (CMU/SEI-2007-TR-002). Software Engineering Institute, Carnegie Mellon University, 2007. http://www.sei.cmu.edu/publications/documents/07.reports/07tr002.html

[Bass 2005]
Bass, L.; Ivers, J.; Klein, M.; & Merson, P. *Reasoning Frameworks* (CMU/SEI-2005-TR-007 ADA441248), Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/publications/documents/05.reports/05tr007.html

[Bass 2003]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice, 2nd ed*. Addison-Wesley, 2003.

[Bellcore 1999]
Bell Communications Research. GR-1400-CORE, *SONET Dual-Fed Unidirectional Path Switched Ring (UPSR) Equipment Generic Criteria*. 1999.

[Bellcore 1998]
Bell Communications Research. GR-1230-CORE, *SONET Bidirectional Line-Switched Ring Equipment Generic Criteria*. 1998.

[Buschmann 1996]
Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns*.Wiley, 1996.

[Dijkstra 1968]
Dijkstra, E.W. "The Structure of the 'THE'-Multiprogramming System." *Communications of the ACM, 11*, 5 (May 1968) 341-346.

[Elnozahy 2002]
Elnozahy, E. N.; Alvisi, L.; Wang, Y.; & Johnson, D. B. "A Survey of Rollback Recovery Protocols in Message-Passing Systems," *ACM Computing Surveys, 34*, 3(Sep. 2002) 375-408.

[Gamma 1995]
Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Garlan 1996]
Garlan, D.; Shaw, M. *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[Gokhale 2005]
Gokhale, S.; Crigler, J.; Farr, W.; & Wallace, D. "System Availability Analysis Considering Hardware/Software Failure Severities." *Proceedings of the 29th Annual IEEE/NASA Software Engineering Workshop (SEW'05),* April 2005, Greenbelt, MD. IEEE, 2005.

[Gray 1993]
Gray, J. & Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[Hamming 1980]
Hamming, R.W. *Coding and Information Theory*, Prentice-Hall, 1980.

[IETF 1981]
Internet Engineering Task Force. *RFC 792, Internet Control Message Protocol*. 1981.

[IETF 2003a]
Internet Engineering Task Force. *RFC 3473, GMPLS RSVP-TE Signaling Extensions*. 2003.

[IETF 2003b]
Internet Engineering Task Force. RFC 3478, *Graceful Restart Mechanism for Label Distribution Protocol*. 2003

[IETF 2003c]
Internet Engineering Task Force. *RFC 5063, Extensions to GMPLS RSVP Graceful Restart*. 2003.

[IETF 2004]
Internet Engineering Task Force. *RFC 3746, Forwarding and Control Element Separation (ForCES) Framework*. 2004.

[IETF 2005]
Internet Engineering Task Force. *RFC 4090, Fast Reroute Extensions to RSVP-TE for LSP Tunnels.* 2005.

[IETF 2006a]
Internet Engineering Task Force. *RFC 4443, Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification*. 2006.

[IETF 2006b]
Internet Engineering Task Force. *RFC 4379, Detecting Multi-Protocol Label Switched (MPLS) Data Plane Failures*. 2006.

[IETF 2007a]
Internet Engineering Task Force *RFC 3623, Graceful OSPF Restart*. 2007.

[IETF 2007b]
Internet Engineering Task Force. *RFC 4724, Graceful Restart Mechanism for BGP*. 2007.

[IETF 2007c]
Internet Engineering Task Force. *RFC 4781, Graceful Restart Mechanism for BGP with MPLS*. 2007.

[IETF 2008]
Internet Engineering Task Force. *RFC 5187, OSPFv3 Graceful Restart*. 2008.

[Jalote 1994]
Jalote, P. *Fault Tolerance in Distributed Systems*. Prentice-Hall, 1994.

[Lyons 1962]
Lyons, R.E. & Vanderkulk, W. "The Use of Triple Modular Redundancy to Improve Computer Reliability." *IBM Journal*, April 1962.

[Mavis 2002]
Mavis, D.G. "Soft Error Rate Mitigation Techniques for Modern Microcircuits." *40th Annual Reliability Physics Symposium Proceedings*. April 2002, Dallas TX. IEEE, 2002.

[Morelos-Zaragoza 2006]
Morelos-Zaragoza, R. H. *The Art of Error Correcting Coding, 2nd ed*. Wiley, 2006

[Musa 1987]
Musa, J.; Iannino, A.; & Okumoto, K. *Software Reliability: Measurement, Prediction, Application.* McGraw-Hill, 1987.

[Powell-Douglas 1999]
Powell-Douglass, B. *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[Saridakis 2002]
Saridakis, T. "A System of Patterns for Fault Tolerance," *7th European Conference on Pattern Languages of Programs (EuroPLoP)*. July 2002, Irsee, Germany. Universitätsverlag Konstanz, 2002.

[Schneier 2001]
Schneier, B. *Applied Cryptography*. Wiley, 1996.

[Scott 2008]
Scott, J. "Evaluating Distributed Systems Architectures for Fault-Tolerant Applications," *Software Engineering Institute SATURN 2008 Conference.* April-May 2008, Pittsburgh, PA. Software Engineering Institute, 2009.

[Telcordia 2000]
Telcordia. *GR-253-CORE, Synchronous Optical Network (SONET) Transport Systems: Common Generic Criteria*. 2000.

[Utas 2005]
Utas, G. *Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems*.Wiley, 2005.

[Von Neumann 1956]
Von Neumann, J. "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components." *Automata Studies,* C.E. Shannon and J. McCarthy, ed. Princeton University Press, 1956.

[Xie 1991]
Xie, M. *Software Reliability Modeling*. World Scientific, 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2009	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Realizing and Refining Architectural Tactics: Availability	FA8721-05-C-0003

6. AUTHOR(S)

James Scott, Rick Kazman

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2009-TR-006

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2009-006

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Architectural tactics are fundamental design decisions. They are the building blocks for both architectural design and analysis. A catalog of architectural tactics has now been in use for several years in academia and industry. This report illustrates the use of this catalog in industrial applications, describing how tactics can be used in both design and analysis. The report further shows how the needs of practice have caused the catalog of availability tactics to be updated, but demonstrates that the underlying structure of the tactics categorization has remained stable. Finally, a real-world example is provided of the application of the updated set of availability tactics, showing how it illuminates design decisions, as guided by associated heuristics and analytic models.

14. SUBJECT TERMS	15. NUMBER OF PAGES
architectural tactic, availability, fault detection tactic, fault recovery tactic, fault prevention tactic	45

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL