Providing Open Architecture High Availability Solutions

Revision 1.0

Published by the HA Forum

February, 2001

Contributing Authors

Tim Anderson, High Availability Software
MontaVista Software, Inc.

Todd Grabbe, Manager, High Availability and Systems Management Development
Radisys Corporation

Julian Hammersley, Systems Architect
Telecom Systems Operation, Hewlett Packard Company

Ira Horden, Staff Systems Architect
Embedded IA Division, Network Processing Group, Intel Corporation

Ken Hosac, VP, Business Development
GoAhead Software

Stuart Levy, Senior Program Manager
Embedded IA Division, Network Processing Group, Intel Corporation

Vince Liggett, Chief Scientist, Telecom Business Unit
Motorola Computer Group, Motorola Corporation

David McKinley, Director of Technology
Computer Platforms Division, Radisys Corporation

David Radecki, Senior Software Engineer
Ziatech, an Intel Company

Brian Ramsey, Director, Communications and High Availability Markets
LynuxWorks

Alan Stone, Manager, Software Architecture
Converged Communications Division, Intel Corporation

Charlene Todd, Staff Systems Architect
Embedded IA Division, Network Processing Group, Intel Corporation

Shinobu "Bob" Togasaki, Chief Architect, Mission Critical Systems
Infrastructure Solutions & Partner Organization, Hewlett Packard Company

Contents

Figures

Tables

1.0 Executive Summary

This document describes the best known methods and capabilities needed for systems that are required to have almost no down-time, frequently referred to as high availability (HA) systems. It is intended as a guide to a common vocabulary and possible HA functions. The system designer must select the appropriate functions for each system based on HA requirements, design complexity and cost.

The document was written by the HA Forum, an industry group with the goal of increasing the number and capability of open architecture high availability systems by standardizing the interfaces and capabilities of building blocks for HA systems. Where blocks are missing the HA Forum will look for solutions to fill the blocks. In addition, the Forum intends to promote solid development models and best known methods for providing high availability systems.

By providing standards and guidelines for open architecture HA systems, development of these systems will become easier. Additionally, systems will be better because developers will be able to focus on their products instead of on low level services required to support high availability.

Three major sections comprise this document: Introductory Material, Functional Capabilities, and Capabilities of Major Building Blocks. A brief summary of each section is given below:

Introductory Material

The introductory material consists of a document introduction, a review of the concepts and principals of developing and maintaining an HA system, and an overview of typical customer requirements for high availability systems.

Concepts and principles of HA are discussed in Section 3.0. In this section, the fundamental principles of engineering HA systems and the issues involved in the design, development and deployment of these systems are discussed. Included are some industry best practices to mitigate the risk and increase the success of deploying HA systems. Some modeling techniques are reviewed to build a quantitative understanding of HA measurement.

Customer Requirements for HA systems are summarized in Section 4.0. This section identifies the application areas under consideration by the Forum. It reviews at a high level the requirements, expectations, and desirable features that typical applications may have of an HA system.

Functional Capabilities

Functional capabilities for HA are separated into those needed to configure and operate the system during normal operation and those that are used to detect and handle faults.

Managing the system configuration is discussed in Section 5.0. This involves knowing what hardware, firmware, and software components are intended to be in the system (the system model) and which of these are actually in a system at any given time. Configuration Management also allows modification of the configuration of the individual components that comprise a system.

Fault management, as discussed in Section 6.0, is typically a five-stage process:

1. Detection – The fault is found
2. Diagnosis – The cause of the fault is determined
3. Isolation – The rest of the system is protected from the fault

4. Recovery – The system is adjusted or re-started so it functions properly
5. Repair - A faulty system component is replaced

Capabilities of Major Building Blocks (or Layers)

In order to create open architecture systems, interoperable building blocks must be available. These blocks can then be combined as needed to create a system. Section 7.0 provides an overview of how a system is divided into building blocks. The blocks are then discussed in terms of hardware blocks, operating system blocks, management middleware blocks and how applications use the blocks.

The capabilities of hardware building blocks are outlined in Section 8.0. Hardware capabilities of fault-managed systems can be generally categorized into three sets: redundancy to allow continued processing after failure, highly reliable (often redundant) communication among components, and management of the components, including fault detection, diagnosis, isolation, recovery, and repair.

Operating system capabilities and qualities needed for HA environments are discussed in Section 9.0. The OS capabilities include functions to isolate, prevent the propagation of, or mask the impact of, hardware and software faults. These functions help prevent errant applications or faulted hardware from bringing the entire system down. Another area of capabilities include dynamic reconfiguration and enhanced device drivers to allow graceful replacement of failed hardware. Additionally, the OS provides services for autonomous fault-management, reporting faults externally, and interfacing with HA capabilities in middleware and application layers.

Management Middleware, covered in Section 10.0, is the software component that oversees all of the configuration and fault management services in the system. The availability management component operates automatically in “real time” and without human intervention. A state-aware system model represents, models and monitors the status, topology and dependencies of components. System information is collected and assessed and system anomalies or faults are detected and diagnosed. Faults are acted upon by dynamically reconfiguring the status, configuration, and dependencies of the components to rapidly recover and maintain service. A final capability of management middleware is that it ***checkpoints*** (periodically transfers) data between a component and its redundant units in order to maintain operation during system reconfiguration.

Applications used in HA systems can either depend on the HA system to simply restart them if a failure occurs, or they can be “HA Aware,” as discussed in Section 10.0. There are many ways for HA aware applications to participate, control, or operate within a highly available system. The system should provide a management interface through which an application can monitor operations and send status, heartbeat and checkpoint information. An application may also need to receive this type of information from other applications. Finally, an application may need to initiate a fail-over or other recovery action and be able to be unloaded, loaded and restarted while the system is operational.

2.0 Introduction

High Availability, or HA, is the term associated with computer systems which exhibit almost no downtime. This document has been generated by the High Availability Forum (***HA Forum***) to make it easier to create open architecture high availability systems using Intel Architecture or other processors. This section discusses the HA Forum, and defines the scope of this document.

2.1 Audience

This document has been written for anyone with an interest in designing and developing HA systems or the hardware or software components used in HA systems. It is also useful as an introduction for purchasers of HA systems. The focus of the document is on the following areas:

- Introduction to HA
- Capabilities used in HA systems
- Implementing HA systems using an open architecture

2.2 HA Forum

The HA Forum was formed to make high availability systems more readily available. The Forum members are Dialogic, an Intel Company, GoAhead Software, Hewlett Packard Corporation, Intel Corporation, LynuxWorks, Inc., MontaVista Software, Motorola Computer Group, Radisys Corporation and Ziatech Corporation.

The goals of the Forum are to simplify and accelerate the development of open architecture systems which can provide high levels of service availability. By establishing standards for these systems it will no longer be necessary for an HA system to be built solely of components from a single company. Open architecture high availability systems will promote more interoperable solutions in the hardware, OS, middleware and application spaces.

2.3 Document Organization and Scope

This document has been written in the order one would follow if designing an HA system. In most cases the following steps would be used:

1. Decide what high availability means for that system
2. Understand what capabilities (and how much of each) are needed
3. Select appropriate building blocks to design the system

Consequently, this document has been separated into four major sections:

- Introduction — Sections 1 and 2
- HA Overview — Sections 3 and 4
- Capabilities to be considered when designing or integrating an HA system — Sections 5 and 6
- Implementation of the capabilities at the various layers in the system — Sections 7 through 11

The scope of this document includes only functions and capabilities specifically related to the key parts of an HA system. These parts are:

1. Redundant hardware and software components
2. Methods for storing information about the components and their relationships
3. Methods for managing faults, from detection through recovery and repair
4. Methods for replacing and upgrading components and updating the stored information
5. Dynamic reconfiguration of the system

Functions normally needed to maintain a secure and well-managed system, such as load management, administration, security (physical and logical), and basic component management, will be included only when needed to explain the main subject.

Topics which are related, but not central to, HA system design will be covered by reference. These topics include such items as:

- Performance management
- Redundant application databases

There are problems that can occur which will make a system unavailable that have more to do with general system practices (administration, security) then they do with HA system design. As such, solving the following problems for the general case is outside the scope of this document:

- Malicious attacks (e.g., “denial of service” network attacks)
- Survivability (from earthquakes, fires, floods, etc.)
- Network congestion
- System overload

3.0 High Availability Concepts and Principles

The demand for increasingly capable hardware and software systems has grown dramatically over the past two decades. Advanced, complex, hardware and software systems have a significant presence in our everyday lives. We often take for granted the tasks and services performed and delivered by our automobiles, telephones, banking institutions, computers, and the Internet. Nearly every industry relies on the availability of large, complex, hardware and software systems to perform important roles that range from enhancing productivity in the work place, to billing customers for services, to ensuring the safety of others, to providing the technology that advances our relentless search for innovation. It seems that when things are running smoothly, we hardly notice these advanced systems in our everyday activities. Yet, when these systems fail to perform their expected functions, they get our immediate attention. A system failure could result in just an inconvenience, but some system failures result in loss of revenue and, at the worst, loss of life.

As our dependency on complex hardware and software systems increases, so does the risk and liability that naturally comes with a potential for failure. The explosive growth of software capabilities in recent years has generally eclipsed the industry’s ability to effectively design, test, and deploy these complex systems to the levels of confidence that consumers are demanding. The problem is further aggravated in software systems because they often also carry the additional burden of responsibility for masking hardware failures.

This section explores the fundamental principles of engineering HA systems and brings an understanding of why the design, development and deployment of highly available systems are such a challenge. This section will also briefly cover some modeling techniques that will help build a quantitative understanding of an otherwise subjective discussion, and finish up with some industry best practices that are used to help mitigate the risk and increase the successfulness of deploying HA systems.

3.1 High Availability and Service Availability

The term “high availability” is frequently used when referring to a system that is capable of providing service most of the time. This is typically quantified in terms of the number of “9s”. Table 1 shows the annual downtime and typical applications for various classes of systems.

Table 1. Classes of High Availability Systems

Number of 9s	Downtime per Year	Typical Application
3 Nines (99.9%)	~9 hours	Typical Desktop or Server
4 Nines (99.99%)	~1 hour	Enterprise Server
5 Nines (99.999%)	~5 minutes	Carrier Class Server
6 Nines (99.9999%)	~31 seconds	Carrier Switch Equipment

Although Table 1 focuses on downtime, the customer or user typically focuses on uptime. This means it is important that a service not only be up except for N minutes a year, but also that the length of outages be short enough, and the frequency of outages be low enough, that the end customer not perceive it as a problem. For most systems, the goal is for few failures and a rapid recovery time. This concept is termed “Service Availability” or making sure that whatever service the user wants (or is paying for) is provided in a way that meets the user’s expectations. Although this is somewhat hard to quantify, the following examples are indicative of what is needed:

Table 2. Service Expectations

Service Type	Service Expectation
Telephone System	Dial tone within seconds of lifting receiver Call completion (ring or busy) within a second of dialing the last digit Call does not get dropped
Web Site	First visual frame within a few seconds
Financial Backend	No lost transactions

In order to provide this type of availability, a system must be designed in a reliable manner, have processes in place to ensure rapid recovery from faults and must be used only within its design parameters. Additionally, the system must be well managed and secure from unauthorized use and activities.

3.2 Terminology

There are many related terms in the context of reliability engineering principles. In order to effectively present and use these terms some definitions are required. Here, we'll apply these terms in the following narration to be able to use each term in a context that is consistent with the goals and positioning of this paper.

A ***system*** is composed of a collection of interacting ***components***. A component may itself be a system, or it may be just a singular component. Components are the result of system decomposition chiefly motivated to aid in the partitioning of complex systems for either technical, or very often, for organizational or business reasons. Decomposition of systems into components is a recursive exercise. A component that is not decomposed further is called an ***atomic component***. Components are typically delineated by the careful specification of their inputs and outputs. A system provides one or more ***services*** to its consumers. A service is the output of a system that meets the specification for which the system was devised, or which agrees with what system users have perceived the correct values to be [Lyu96].

A ***failure*** in a system occurs when the consumer (human or non-human) of a service is affected by the fact that the system has not delivered the expected service. Failures are incorrect results with respect to a specification or unexpected behavior perceived by the consumer or user of a service. The cause of a failure is said to be a ***fault***. Faults are identified or detected in some manner either by the system or by its users. Finally, an ***error*** is a discrepancy between the computed, measured, or observed value or condition and the correct, or specified value or condition. Errors are often the result of exceptional conditions or unexpected interference. If an error occurs, then a failure of some form has occurred. Similarly, if a failure has occurred, then an error of some form has occurred. Since the difference between an error and a failure is very subtle, the remainder of this document will treat the terms synonymously.

Note: Faults are ***active*** when they produce an error. Faults may also be ***latent***. A latent fault is one that has not yet been detected.

Since faults, when activated, cause errors, it is important to detect not only active faults, but also the latent faults. Finding and removing these faults before they become active leads to less downtime and higher availability.

System designers often build reliability into their platforms by building in correction mechanisms for latent faults that concern them. These faults, when correctable, do not produce errors or a failures since they are part of the design margins built into the system. They should still be monitored to measure their occurrence relative to designers anticipated frequency, since excessive occurrence of some correctable faults is often an indicator of a more catastrophic underlining latent fault. For example voltage fluctuations that are inducing correctable data path errors.

Dependability is defined as the trustworthiness of a system such that reliance can justifiably be placed on the service it delivers [Lapr92]. Dependability has several attributes. ***Availability*** is defined as the readiness for usage. The continuation of service, in the absence of failure, is called ***reliability***. The nonoccurrence of catastrophic consequences or injury to the environment or its users is called ***safety***. The nonoccurrence of unauthorized disclosure of information results in ***confidentiality***. The nonoccurrence of improper alteration of information results in ***integrity***. The ability to undergo repairs and evolution provides ***maintainability***. And finally, the association of integrity, and confidentiality, results in ***security***.

When we speak of reliability, it is typically quantified by its failure rate, or ***mean time to failure***, or ***MTTF***. This attribute is the interval in which the system or element can provide service without failure. It is represented as a reciprocal of the statistical mean elapsed time to its projected or observed failure. Another attribute related to reliability is the ***mean time to repair,*** or ***MTTR***. This attribute represents the interval in time it takes to resume service after a failure has been experienced. The availability is then expressed by:

Equation 1.

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

As you can derive from this very simple formula, the secret to high availability is either creating very reliable elements (very high MTTFs) or creating elements that can recover from failure very rapidly (very low MTTRs).

3.3 System Reliability and Availability

3.3.1 Reliability vs. Availability

As introduced earlier, reliability is a dependability attribute. It is a measure of the continuous delivery of a service in the absence of failure. Reliability is most often represented as a probabilistic number or formula that estimates the average time until failure, or MTTF. By definition, the use of this measure is of a limited confidence since it is probabilistic.

Availability, another attribute of dependability, is a quite different measure. It is the measure of the probability that a service is available for use at any given instant (and potentially in turn for some interesting interval thereafter). Availability allows for service failure, with the presumption that service restoration is imminent. The key to high availability is to minimize the restoration intervals. In Equation 1, as the MTTR approaches zero the availability approaches 1, or 100%.

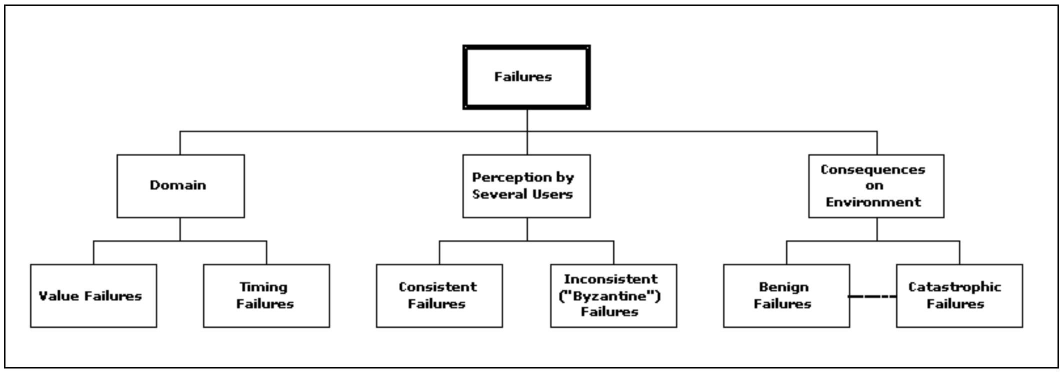
The design and development of highly reliable systems is very challenging. Because the techniques used to design, develop, and test systems with high reliability goals are typically very expensive, building highly reliable systems is often limited to special industries and applications. Such is the case in many avionics, life-support, military, and aerospace programs. In many of these environments, the presence of a failure is often deemed potentially life threatening; hence, every precaution is taken to build reliable systems because they cannot tolerate the repair intervals of failures. For most other applications, however, the ability to provide near continuous service by repairing faults and preventing their propagation is more economical and still acceptable by its users.

Even when building to the toughest requirements for extra-high reliability, one has to consider the presence, containment, and restoration of service due to failure. Most engineers share the opinion that non-faulty systems do not exist; there are only those systems that have not yet failed.

3.3.2 What Compromises a System’s Reliability

Faults, errors, and failures are the conditions that ultimately compromise a system’s reliability and in turn its availability. Remembering that a failure is a reflection of an unacceptable or incorrect result delivered by a system as perceived by its user(s), failures can be classified into three viewpoints [Lapr92] as shown in the following figures.

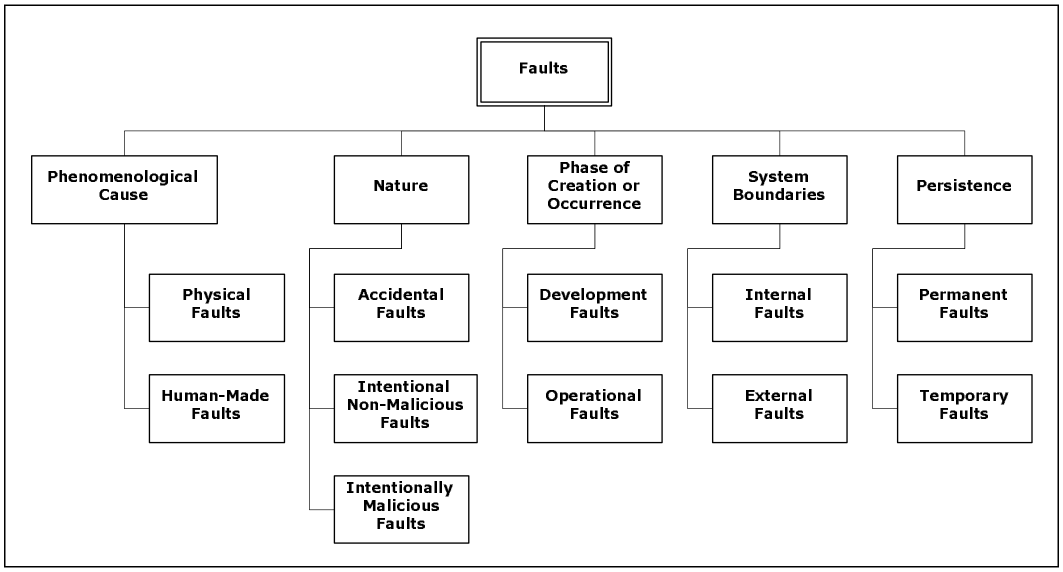
Figure 1. Failure Classes



These failure classes are a generalization of the ways in which a system may fail. Failures that are related to value and timing failures are regarded as halting failures. They reflect the absence of activity from the system and are often caused for example by a failed component. Other failures, often the result of development faults, manifest themselves as either consistent failures, or as Byzantine failures. That is, they are often easily reproducible or inconsistent, the latter typically being caused by latent faults. A system that generally only exhibits benign failures is sometimes referred to as a fail-safe system.

Faults and their sources are varied and diverse. In ‘Dependability: Basic Concepts and Terminology, Dependable Computing and Fault Tolerant Systems,’ [Lapr92] Laprie classifies them according to five main viewpoints shown in Figure 2.

Figure 2. Fault Classes



This classification is useful in understanding the domain of faults one needs to protect a system against. It is interesting to point out that there is a wide range of human-made faults. For example, human-made faults may be of non-malicious or malicious intent, and it is the result of these faults that are often the chief contributors to a system compromised reliability. An example of non-malicious intent might quite simply be an error in the design or implementation of a component in the system. Similarly, systems may be compromised intentionally, but still non-maliciously, perhaps because of design tradeoffs, where certain limits were intentionally disregarded because of the infinitesimal probability of their existence. At the other extreme, human-made faults may intentionally, maliciously, compromise the reliability of a system as it so commonly demonstrated in this Internet era through the use of viruses or Trojan horses.

A key to increasing a system's availability is found in seeking defense from these impairments. In the following section some remedies are explored.

3.3.3 Faults — Prevention, Removal, Tolerance, and Forecasting

When designing and developing a highly available system it is critical that faults be minimized and procedures be put in place to handle them when they do occur during operation.

The following four primary methods aid in the design and development of reliable systems; fault prevention, fault removal, fault tolerance, and fault and failure forecasting. These processes are used in the development phase of the project. In later sections we will discuss how to handle faults once the system is in operation.

Fault Prevention

Fault prevention, while not considered by [Lapr92] and [Rand95], is considered by [Lyu96]. The application of best-known methods for design and development, the preference for simplicity over complexity, the refinement of user requirements, and the discipline for the execution of sound engineering practices are perhaps the best defense against faults being created in systems. Formal methods have been researched, but are primarily are still left to academia. Few formal methods are routinely practiced in the industry.

To help combat the significant influence of software failures in large systems, software reuse can be applied. The continual improvement in reliability when reusing software has demonstrated benefits. This concept is known as reliability growth. Object-oriented software engineering practices further encourage software reuse and have also demonstrated significant improvements and reliability growth when maintained and reused.

Fault Removal

Fault removal during development is comprised of three steps: verification, diagnosis, and correction. ***Verification*** is the process of checking if the system is fulfilling its intended function. If it does not fulfill the intended function, ***diagnosis*** and subsequent ***correction*** is required to remove the fault and failure condition. Verification techniques vary but generally may be characterized as being either static or dynamic.

Static verification involves the static analysis of a system. This may be the application of design inspections, data flow analysis, quantitative or proof-of-correctness analysis. In each of these cases the system is verified (at least to some extent) without its execution or delivery of service. Dynamic verification requires the system to be executing its function(s) with the intent to verify the mapping of input stimuli to output response. In most large complex systems, the ability to verify all possible inputs and their representative outputs is typically an intractable problem.

The emphasis of fault removal is different for hardware and software systems. In hardware systems the emphasis is on the removal (or more commonly referred to as the *repair*) of potential production faults. In software, the emphasis is on the removal of design faults.

When fault removal is considered after a system has been commissioned, it is called ***corrective maintenance*** [Lapr92]. This is either ***preventative maintenance*** to ward off faults before they produce errors, or ***curative maintenance***, which is aimed towards the removal of faults reported against the system.

Fault Tolerance

Fault tolerance is an attribute associated with a computing system that provides continuous service in the presence of faults. The active ability of a system to circumvent or otherwise compensate for activated faults is varied among different fault tolerance techniques. In the case of hardware, fault tolerance is often achieved through the use of redundant elements in the system. In the case of software, fault tolerance techniques generally are realized through design diversity. Multiple implementations of software provide the same purpose. The implementations are often constructed by different development teams, designing to the same specifications but with alternate implementations. Two popular approaches include N-versioning and the recovery block method.

Redundancy may also be applied to both software and hardware systems, but it is typically less effective in software systems. Redundancy in software can help the continuity of services that exhibit long degradation intervals. This is often how bugs pass through verification testing and make their way to the field. In these cases, the faults usually only manifest themselves under certain timing and loading situations (that inevitably occur at key customer sites) that are too stochastic to be reproducible. These bugs are often referred to as Heisenbugs [Gray92]. The causes of Heisenbugs are seldom ever determined. In these cases, software redundancy can help as the long term timing of events in the software modules will seldom be exactly the same. The differences in timing can be increased by periodically causing a switchover between the modules and restarting the no longer active module.

Fault and Failure Forecasting

The ability to manage reliability futures is an instrumental part of the complete lifecycle of system availability. Understanding the operational environment, gathering field failure data, the use of reliability models and the analysis and interpretation of these results are all significant and important to successfully manage availability.

Fault and failure forecasting includes the understanding of a related term called reliability growth. Reliability growth is defined as the continued improvement of reliability in systems. It is generally measured and achieved by the successive increase in intervals between failures. By using and applying the methods described in this section, reliability is expected to improve over time, and improve between system versions and variants when these methods are reused and matured.

3.3.4 The Challenges of Making Highly Reliable and Highly Available Systems

When designing a system, requirements and a great deal of engineering typically go into optimizing three primary dimensions; cost, performance, and dependability. The maturity, accuracy, and repeatability of the cost and performance dimensions are well understood and demonstrated in most industry areas. However, the ability to understand and demonstrate the dependability of systems is generally lagging in most industries.

All of the methods described to this point are goals to strive for when designing systems for high availability. Unfortunately, each of these goals, techniques and methods are subject to imperfections. Contamination occurs most often through human error. Either in the design, implementation, or use of systems, faults are inevitably created.

The complexity of modern systems has increased so significantly in recent years that it is rare that a single company can provide all components of the system. Therefore, ***Commercial Off The Shelf*** (***COTS)*** components are used to help integrate functions that are not the core competency of the company developing the system. These components, when stitched together, ultimately yield a web of complexity in their dependencies. Often, the low level of characterization that accompanies COTS components aggravates issues associated with building dependable systems. Hardware components generally have a much stronger discipline in their characterization than do software components. It is this inherent use of these components and their coupled errors that requires the use of fault removal in systems in order to maintain high levels of availability.

The fact that software reliability engineering has not advanced as quickly as hardware reliability engineering is further aggravated by the fact software is often used to mask hardware failures. In many systems, user interfaces are either controlled by or presented by software; hence, they are the barrier between the user and a system exposing failures via the loss of service or incorrect service from those interfaces. Oftentimes, these software systems control and manage the loose coupling between user and system, and when possible, provide failure avoidance measures.

Software reliability is similar to hardware reliability in that both are stochastic processes and can be described by probability distributions. However, software reliability is different from hardware reliability in that software does not wear out, burn out, or deteriorate (i.e., its reliability does not decrease with time). Additionally, software generally benefits from reliability growth during testing and operation since software faults can be detected and removed when software failures occur. On the other hand, software may experience reliability decrease due to abrupt changes of its operation usage or incorrect modifications to the software. Software is also continuously modified throughout its life cycle. This malleability of software only increases the risk of introducing new errors.

Unlike hardware faults that are mostly physical faults, software faults are design faults, which are harder to visualize, classify, detect, and correct. As a result, software reliability is more difficult to realize and analyze than hardware reliability. Usually, hardware reliability theory relies on the analysis of stationary processes, because only physical faults are considered. However, with the increase of system's complexity and the introduction of design faults in software, reliability theory based on stationary process becomes unsuitable to address non-stationary phenomena, such as reliability growth or reliability decrease that is experienced in software. This makes software reliability a challenging problem that requires several methods of attack.

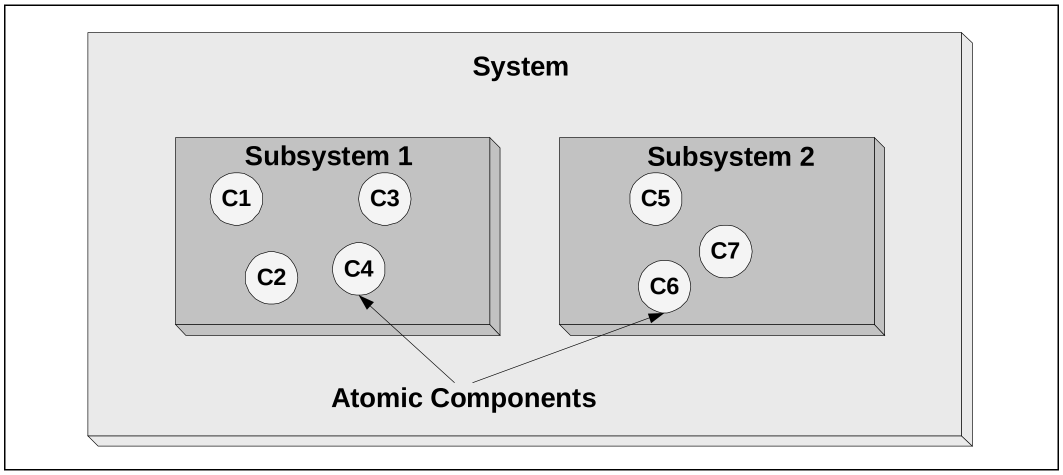
3.4 Reliability Modeling for Hardware and Software Systems

In this section, a unified view of modeling hardware and software is presented. Introductory reliability theory is presented as it applies to this view of a system model. The motivation and consideration for presenting interacting components is reinforced through the introduction of mathematical modeling. An analytical view of reliability is presented, system availability when service restoration is considered, and reliability forecasting is briefly introduced.

3.4.1 System Decomposition

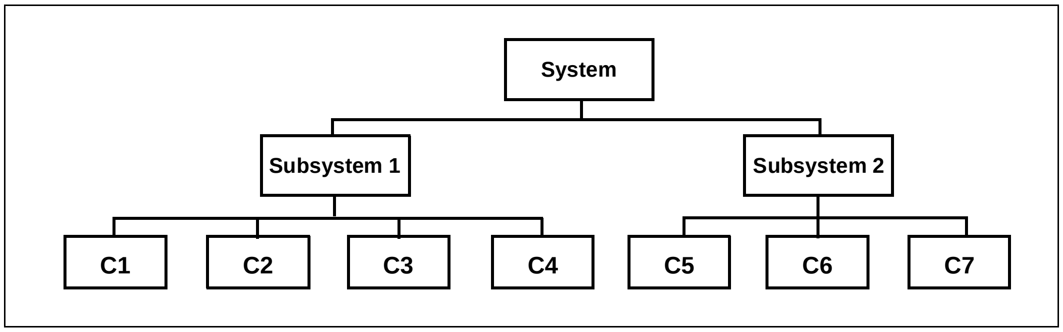
Earlier, a system was defined as a collection of interacting components, which, in turn, may again be comprised of components. This recursion is ultimately terminated when the system has been decomposed to the point where further decomposition is of little value or interest and a matter of diminishing return. The smallest units of system elements then considered are called atomic components.

Figure 3. System Decomposition



This view of system decomposition is useful in representing the "composed-of" relationship of components in a system. An alternative view could be shown as a tree as depicted below.

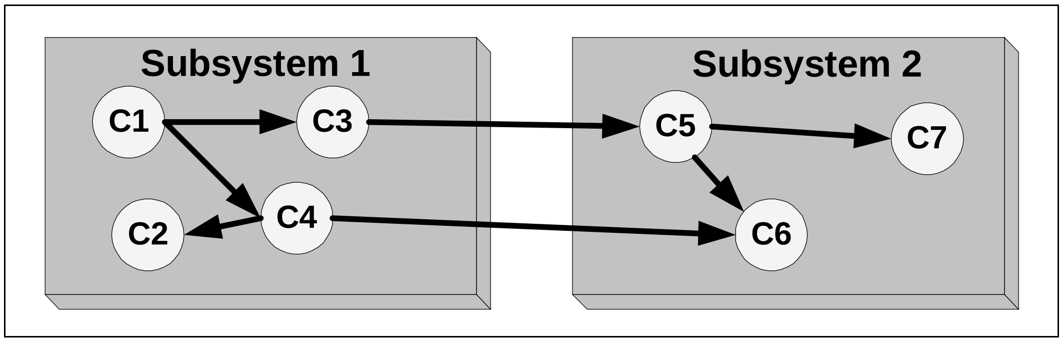
Figure 4. Tree View of System Decomposition



This view implies a hierarchy that describes the composed-of relationship between a system, its components, and their components. More levels in the hierarchy provide more detail, and hence, smaller components are present in the system model.

While this depiction is useful for the understanding of the hierarchical aggregation of components that comprise a system, the understanding of the relationships between components is not revealed. A system view including relationship information is critical to the understanding of the influence one component has on another. A component has a specific relationship with other components in the system. Components may supply a service, or may consume a service (it may do both as well) from another component. Overall, in the macro view of our system, the system provides one or more services to its consumers (users). To better depict this relational context one can employ a "uses" diagram instead of a "composed of" diagram, as illustrated in Figure 5.

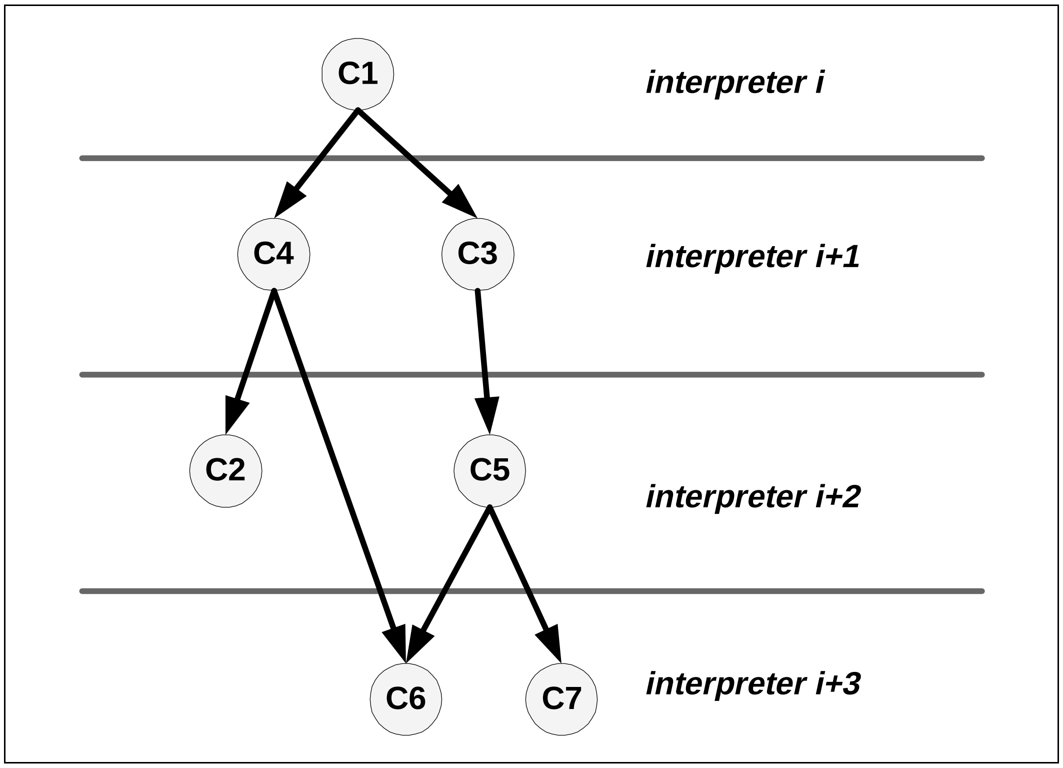
Figure 5. Relational Component Diagram



In this figure, the dependency of one component on another is shown. For example, C1 uses the services provided by C3 and C4. Note that C4 depends on the services of C2 and the components referred to here are either hardware or software components.

Finally, another view of a system is considered. The relational 'uses' view described above is often lost when a system is realized. For example, consider the case of software components. The components are a logical grouping of source code, but when compiled by a compiler, these artificial boundaries are destroyed and a more monolithic product is produced. Even when writing layered software, the layers are dissolved away after the compiler creates machine code instructions. Components and layers are convenient and effective ways for managing the source code of software but are not representative of the running system that needs to be modeled. In 'Fault Tolerance — Principles and Practice,' Anderson introduces another type of relation called the *interpretive interface* [Ande81]. It suggests that a system may be viewed as a hierarchy of interpreters, where a given interpreter provides a collection of abstract objects to an interpreter above it. For example, a collection of hardware components that comprise a computer motherboard *interprets* the application that is hosted on it. The interprets view is depicted in Figure 6.

Figure 6. The Interpretive Interface



In Figure 6, the components have been stratified to show their interpretive relationships. Components C6 and C7 might represent hardware components, and C2 and C5 might represent operating system components, for example. This view will be used in the following section to help derive failure rates of systems modeled this way.

Section 5.0 uses these concepts when defining a system model. The system model is a computer-usable representation of the capabilities, characteristics and dependencies of all of the components that could be included in a system.

3.4.2 System Models without Service Restoration

In this section, the reliability of systems without service restoration is examined. This will provide an understanding of reliability in the absence of fault recovery, providing for example, the mean time-to-failure for a collection of interacting components. Here, relevant summarized citations are made from [Lyu96].

The simplest approximation of a system's failure rate, λ, can be shown to be:

Equation 2.

$$\lambda = \sum_{i=1}^{C} \pi_i \lambda_i$$

Where C is the number of components in the system, and π is the average proportion of time component i is under execution. It is the sum of each of the components failure rates, with a compensator for its actual sojourn time.

Equation 2 may be used for both hardware and software system failure rate analysis. When just hardware components are considered it is generally accepted that all components are running all the time. All the π_i terms equal 1 and Equation 2 simply becomes the sum of the failure rates of each of the components, shown in Equation 3.

Equation 3.

$$\lambda = \sum_{i=1}^{C} \lambda_i$$

For software systems and their components, the most challenging problem is how to estimate the failure rates of each of the components.

In the previous section, the interaction between components was considered with an *interpretive* view. Revisiting this model [Lyu96] shows that the failure rate at any of the interpreters, or components can be derived via some vector arithmetic. Perhaps most importantly, the failure rate of the system, also considered being the first layer interpreter, can be derived. In the simplest case of a single software interpreter and a single hardware interpreter, and using the same assumptions made above with regard to hardware components always running, the system failure rate can then be approximated by:

Equation 4.

$$\lambda = \sum_{i=1}^{C_S} \pi_i \lambda_i + \sum_{j=1}^{C_H} \lambda_j$$

Where i is an iterator across all of the software components and j is an iterator across all the hardware components.

3.4.3 System Models with Service Restoration

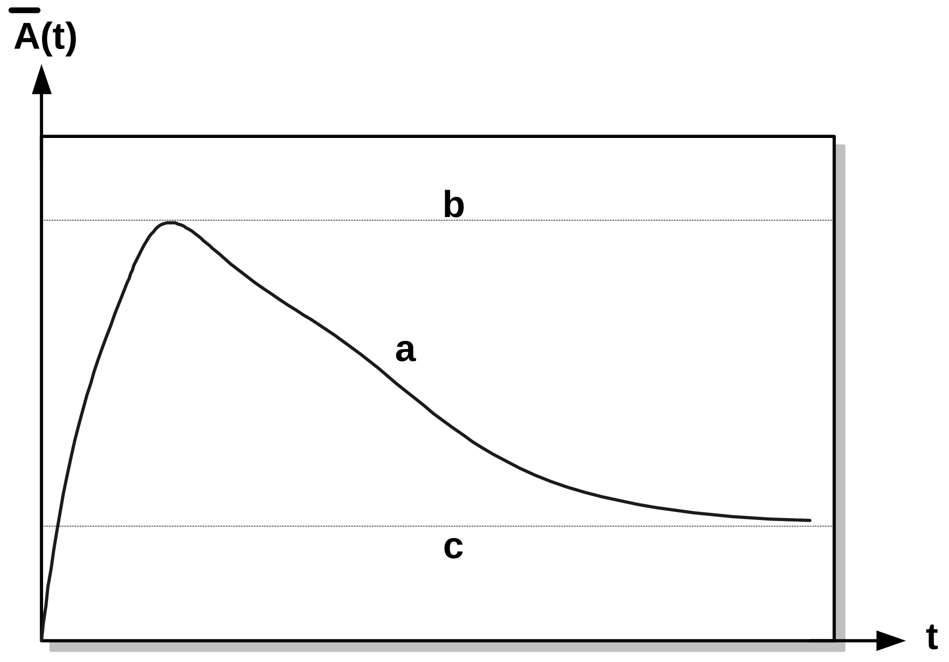
Section 3.4.2 addressed systems in the absence of service restoration capabilities. The behavior of a system when failures are removed and service is restored was not taken into account. The understanding of this principle is important to better understand how to model the availability of a system.

In this section, the influence service restoration has on the behavior of a system will be addressed. Restoration may manifest itself in several ways. It may be as simple as restarting the failed component. It may mean that the component needs to be replaced by an identical one. It could also mean that the component has a newly identified defect and that it must be replaced by a new version.

Service interruption can be qualified by at least two important attributes that directly affect the availability of a system: The time to restart a system (or component) after a failure, and the time to restart a system (or component) after the introduction of a new version of that system (or component). These attributes are good elements of measure. Oftentimes, the introduction of new versions is driven by the failure intensities that have been experienced so far. However, the introduction of new versions may be the cause of potentially new faults. Hence, risk evaluation is required when considering a version upgrade.

Systems may be characterized by their failure intensity over time. Generally, newly deployed systems exhibit higher failure rates with defect removal occurring over the near term as a system is deployed into new environments. These residual design faults are common in new systems. Their removal is a trend of decreasing failure intensity. Eventually, systems generally achieve a stabilization of reliability over time when changes to the system and environment are no longer common. The failure intensity increases as changes to the system or environment happen (e.g., new versions of hardware and/or software are added to the system). Over time, the unavailability (A) curve looks something like Figure 7 [Lapr92]:

Figure 7. Unavailability Curves



The trend of the ‘a’ line depicts the results of a hyper-exponential model [lapr92] that is also backed by typical field data. It shows that systems typically peak in their unavailability shortly after deployment, then, through reliability growth (defect removal), eventually stabilize near their expected reliability level (c). The ‘b’ line represents the pessimistic reliability prior to defect removal and reliability growth. The ‘c’ line represents the optimistic reliability after reliability growth has taken place.

The ability to model the reliability and availability of a system taking into account reliability growth is a relatively complex topic beyond the scope of this paper, however they can be approximated by a statistical estimation. Details on this can be found in Laprie’s ‘Dependability: Basic Concepts and Terminology [Lapr92].

In that example, the availability of a system is given by the ratio of non-failed components at time t to total number of components in the set. Integrated over time, the average gives an approximation of the availability of the system from its time origin. The intervals or quantization of t over which the availability is measured sets the granularity of observed availability. This estimation is a passive appraisal of availability. That is, it is a simple way of expressing the observed availability of a system once it is commissioned. An active model, like the one suggested by Laprie, offers the ability to analytically estimate system availability prior to its commissioning [Lapr92]. While this is only a model, it is one of the responsible ways in which system engineers can increase their confidence and help reduce the risks in their ability to design and deploy highly-available systems.

3.5 Redundancy

Almost all hardware fault management techniques utilize redundancy, so it is helpful to expand this further. Redundancy is generally classified into three basic types:

1. Spatial, which consumes space. This involves provisioning of more resources to provide a service than would otherwise be needed – the extra resources are used when a primary resource fails.
2. Temporal, which consumes time. An example of temporal redundancy is the ACK/NAK method used by many protocols. A failure of reception causes the message to be repeated, which is the redundancy, while the penalty is that of time.
3. Structural, sometimes called contextual. Here the redundancy is held at somewhat less than a full duplication by making use of properties of the data. Examples are the Hamming codes used in error correction, or the exclusive-OR commutative property used by RAID subsystems.

To be effective, redundancy must be applied in such a way that one fault could not cause both copies of a component to fail. This brings in the notion of a ***fault domain***, which can be defined as the locus or scope of a fault. By recognizing that faults have a defined locality or domain, then it is a necessary condition for fault tolerance that redundant components must be provisioned in different fault domains.

To illustrate this, consider a system having a bus, in which a peripheral adaptor provides a service. Simply replicating the adaptor on the bus is not enough to ensure continuity of service, since a bus fault can cause both adaptors to be unreachable. The adaptors are replicated within the same fault domain – that of the single bus. This is why bus-based highly-available systems contain at least two buses, with redundant components allocated to each bus.

One or more primary components, together with their redundant counterparts, act together to provide a reliable service. A group of such components is defined as a ***service group***, an example of which might be the power supplies in a chassis, assuming that they were configured such that at least one was redundant.

Spatial redundancy can be applied in a number of different ways as described in the next four sections.

3.5.1 Classical Fault Tolerance

Redundant components are used to process identical data, and a voting process ensures identical results. Of course, since the voter itself could be erroneous, then that also must be redundant, and there are various schemes possible.

Two such service-providing components, along with their associated voters, are known as ***dual modular redundant*** systems, or ***DMR***. While DMR systems provide the basic integrity checking, in the event of a voting disagreement it can be difficult to determine the unit at fault. Accordingly, it is more common to employ such components in groups of three, which makes possible the “odd man out” process of fault diagnosis. Such triplicated components and their voters are known as ***triple modular redundant***, or ***TMR***. The provision of hardware and voters three times has a very adverse impact on cost.

3.5.2 Standby, or Hot Sparing

A more basic method of providing redundancy is simply to add an additional component to the system. This acts as a standby component and is not used until the primary service provider fails. If the primary service provider is disabled, the standby component becomes active, restoring the service. Such simple duplication is frequently known as ***active/passive***, or 1+1 redundancy. Determining which adaptors are paired, and which are active or standby, is known as role assignment, and is a necessary task for such systems.

An extension of this is where a service is provided by a number of identical components such as line adaptors. Here it is possible to over-provision the line adaptors and designate the excess as standbys. The general case is referred to as ***N+M redundancy***, where N is the number of primary service providers, and M is the number of stand-bys.

3.5.3 Load Sharing

Rather than having the redundant hardware be idle, it is more cost-effective to allow the back-up hardware to share in providing service. This is usually more difficult to manage, since a function to distribute the tasks between the service providers must be provided, and furthermore the service performance may be seen to degrade when one of the providing components fails. Typically, a minimum set K of providers is defined which will satisfy the service requirement, and then the load is further spread over the full set N of providers. This is designated as K:N redundancy (pronounced K out of N) and has a minimum value of 1:2.

3.5.4 Clustering

The above sections refer to components that provide service, and can be applied no matter what the size, scope, or nature of the component. There is however, a special case where the service-providing component in question is itself a complete computer system, containing hardware, operating system, and communications capabilities. Such a component is known as a ***node***, and a system constructed using such nodes is known as a cluster.

Clusters can be homogeneous, when they are composed of identical nodes, or heterogeneous, when the nodes can vary widely in make-up or even architecture. Nodes are managed on a black-box basis – either the node is fully functional, or the entire node is taken out of service with no attempt to diagnose or rectify failures within the node. Failures therefore remove more of the system (i.e., the whole node) instead of the single failed component.

Nodes are linked by a network, which must of course be reliable and is therefore frequently redundant. Static role assignment can also be performed on the nodes making up the cluster system; however, more advanced systems can employ smart network switches and end-points to provide a dynamic reconfiguration capability which can enable a more extensive coverage in the presence of multiple faults.

Services provided by a node in a cluster can therefore migrate to another node in the event that the primary node fails. Services therefore need to be location independent, and various software mechanisms such as CORBA* are available to implement this.

The goal of cluster technology is to improve service availability. However, depending upon industry and the application, the term cluster may describes different implementations. The following definitions of clusters are used:

E-Commerce Cluster: Clustering is an commercial term describing the technology that allows multiple nodes to be deployed as a single network-attached computing resource. In this implementation, complete systems are redundant, and the redundancy is completely transparent to the consumer of the service. The HA capabilities of this type of cluster include failover modes, management APIs, storage integrity, predictive failover, arbitration and recovery. This cluster technology is commonly deployed in e-commerce and IT applications [DHB3’00].

Network Backplane Cluster: Another approach to high availability and live insertion is to implement a system that uses a dedicated network fabric that is separate from the backplane. In this implementation hardware components are intelligent network nodes, not I/O cards. As in a backplane system, the system may be designed with complete n+m redundancy. In order to eliminate a single point of failure, two redundant networks connect each node. Because of modern network design, a node failure will not bring the network down, and the units can be replaced or reprovisioned without bringing the network down or affecting the service capability of the system. In this implementation, the failure of a single critical I/O channel on a node will fault only that node. Network technologies typically used for this type of implementation include 100/1000BASE Ethernet, FDDI, token ring and ATM. Switched Ethernet has the advantage of point-to-point bandwidth and the promise of QoS. Token Ring and FDDI provide assurance of worst-case delivery time. ATM offers high bandwidth and QoS.

Element Cluster: Telecom Equipment Manufacturers typically use cluster as the description of all of the systems that provide the service function of a network element. The elements of the functional cluster may be heterogeneous systems, and x-depth duplication of homogenous systems to provide element scalability. Any of these classes of systems, can in themselves be a network or e-commerce system. For instance, a MTS element may include high availability switching elements as well as e-commerce-like redundant clusters to maintain copies of home records and to ensure local, secure storage of billing information.

3.6 Making it All Work — Open vs. Proprietary

The overarching goal of this high availability framework is to provide an open system environment in which multiple vendors can participate in providing system services that help achieve the expected level of availability in the systems we design. Today, systems are far too large and complex to expect that a single vendor can provide all of the functionality required of a system. Inevitably, you will depend on the services of other vendors to realize your system. By providing an open system framework, we can expect the dynamics of an industry to align with an initiative to provide interoperable services that can participate in a controlled and uniform manner. This is what is alternatively achieved if you were to do it all yourself (or via contract) in a proprietary system. The uniformity gained by providing a common framework provides a significant reduction in complexity of otherwise integrating disparate components and services.

By providing an open system approach to building high availability systems we can expect that all parties can benefit from the reduced deployment impedance of providing HA systems. Faster time-to-market can be realized via the use of widely available and reusable services that participate in a framework that will encourage the maturity and growth of system services providing increased reliability. Through such a framework, it is expected that the participation of such components and services will provide the ability to scale or grade the performance and reliability to meet the needs of the system designer. By formally managing the participation of these otherwise disparate technologies from multiple vendors, we can expect to see improved reliability growth as they participate in a prescribed manner, limiting the usage profile to a tractable set of use cases. This gives a higher level of confidence in the validation tests that qualify these services before their deployment. This will result in fewer faults in the field, providing more profitable products, and much happier consumers.

4.0 Customer Requirements for Open HA Systems

For many years computers have been controlling systems that provide critical services where continuous availability and data integrity are essential. The scale, the availability requirements, and the level of data integrity vary widely across the spectrum of such services. Consequently, the techniques and implementations that provide the high availability (HA) are also equally varied. However, the history and evolution of these applications has added extra dimensions of complexity and variance to the HA implementations. Often the development has been ad hoc and narrowly focused, and has usually led to proprietary solutions that are fragile and expensive to support. Much of the same technology has been re-invented time and time again, but in ways that are incompatible and impossible to reuse.

With the widespread adoption of open systems and more horizontally-focused component suppliers, there is growing expectation for an open system approach to provide products and services that address the needs of high availability in a variety of application spaces.

An Open System HA Framework Can:

- Standardize the design of commonly used HA components and techniques
- Encourage the development of Commercial-Off-The-Shelf (COTS) HA components
- Allow competitive environment to improve product features, cost, and performance
- Allow application developers to focus on core value-add development
- Reduce the HA application development time and cost, and improve its supportability
- Improve the portability of HA applications between hardware platforms, operating systems and middleware
- Improve the overall reliability of systems and applications by leveraging the HA component testing over a wider installed base
- Provide a consistent basis for comparing the HA features offered in different systems
- Improve the supply of staff with relevant HA technology experience

When formulating the direction and definition of such an open systems approach, it is important to consider the range of applications, and the scope of their requirements. This section identifies some of the application areas considered for an open systems HA framework architecture, and reviews at a high level the requirements, expectations, and desirable features that typical applications may have for an HA framework.

4.1 Application Areas

While the primary application area for an open HA framework is broadly aimed at the systems that provide telecommunications and Internet infrastructure, many other application areas share similar types of architectures and requirements, and will also benefit from an open HA framework. Some examples of telecommunications and Internet infrastructure applications include:

- SoftSwitches and VoIP gateways
- Telecom network service control points (SCP)
- Telecom network switch transfer Points (STP)
- Wireless base station controller (BSC), radio network controllers (RNC)
- Remote access controllers
- Web hosting, caching, and email servers
- Broadband distribution nodes
- Voice portals and unified messaging systems

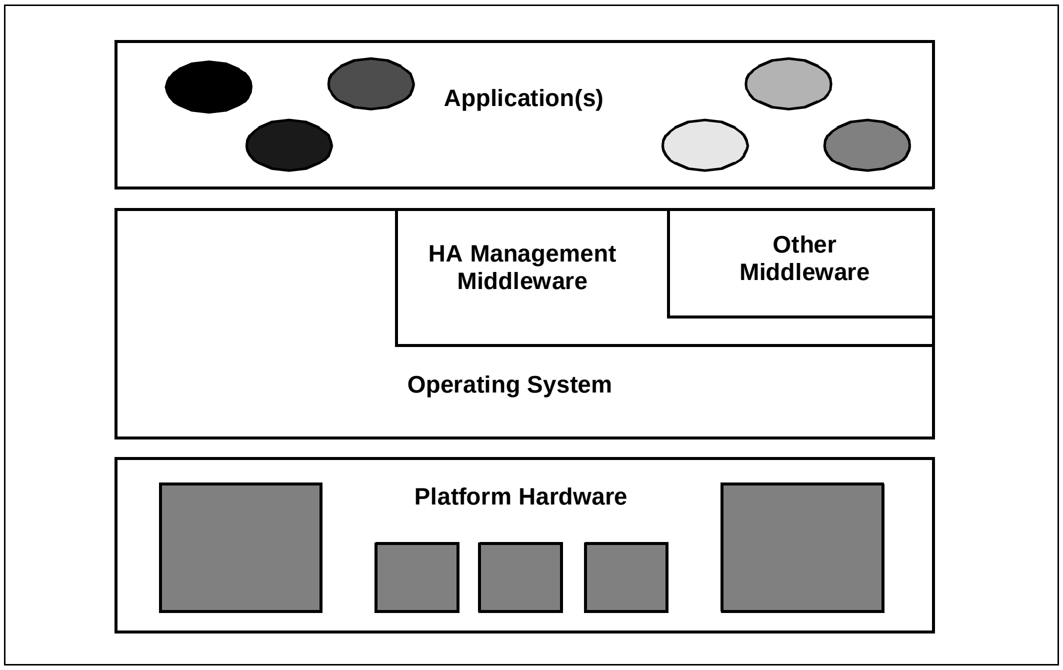
These systems are components in the larger network infrastructure and provide critical services in the data path, control nodes, network management elements, as well as the billing and the application services areas. There is growing reliance on this infrastructure with the consequent expectation of its continuous availability. The availability expectation also assumes that the desired levels of data integrity, performance, Quality of Service (QoS), and security are acceptably maintained.

4.2 Open HA Framework Outline

The primary goal of the HA Forum is to describe an Open HA Framework. Since the idea of an open HA framework is that systems can be built of compatible modular components - the Framework must describe and bound these components, and define the standard interfaces between them. It must not only describe the modules and interfaces within an individual system, but also the interfaces between the systems.

The framework must be able to be implemented in a heterogeneous hardware and software environment and be compatible with the features and capabilities of existing open hardware and software standards. It must also consider the design methodologies used in existing HA applications to ease the migration and porting effort of such applications to the open systems HA approach.

Figure 8. Open HA Framework Individual System Model



The framework should also provide some design guidelines for good HA implementations for both hardware and software within an open HA framework environment.

4.2.1 Scope of an Open HA Framework

The design of an open HA framework needs to consider the full environment and full life-cycle of components in the target infrastructure. These include:

- Compatibility with existing and legacy systems and infrastructure
- Design, porting and troubleshooting of applications
- Installation into the infrastructure
- Manageability and security
- Performance, cost and scalability
- Component failure, isolation and recovery
- Testing and repair
- Hardware and software changes and upgrades

4.2.2 Compatibility and Interoperability

The definitions of an HA Framework must be clear and unambiguous. They must define the component interfaces well enough to ensure that different implementations of the same components will interoperate.

The definitions may leave room for future expandability and growth, but must require standards-compliant implementations to be compatible. While it is possible that the component boundaries may change over time, it is important to clearly define the initial models and implementations.

4.2.3 Related Standards Considered

The HA Framework must leverage or recommend existing hardware and software standards and describe how they are used and integrated into the HA Framework standards. Some may be considered prerequisites and listed as requirements, while others may emerge as the HA framework definition evolves. For example:

- Management: WBEM, SNMP, IPMI, etc.
- Configuration/system modeling: CIM, X.731, etc.
- I/O standards: PCI, CompactPCI*, H.100/110, InfiniBand*, etc.
- Networking standards: TCP/IP, ATM, T1/E1, Sonet/SDH, etc.
- Middleware standards: CORBA, etc.
- De-facto standards for operating system, middleware etc.

4.3 System Topologies and Components

Typically the telecom and Internet infrastructure applications are built using clusters of systems connected by network(s) such as SAN(s) or LAN(s). These clusters contain extra components (e.g., redundancy organized in modes such as N+1, 2N, etc.) that can be reconfigured in the event of a single component failure so that the overall service provided by the cluster is acceptably maintained.

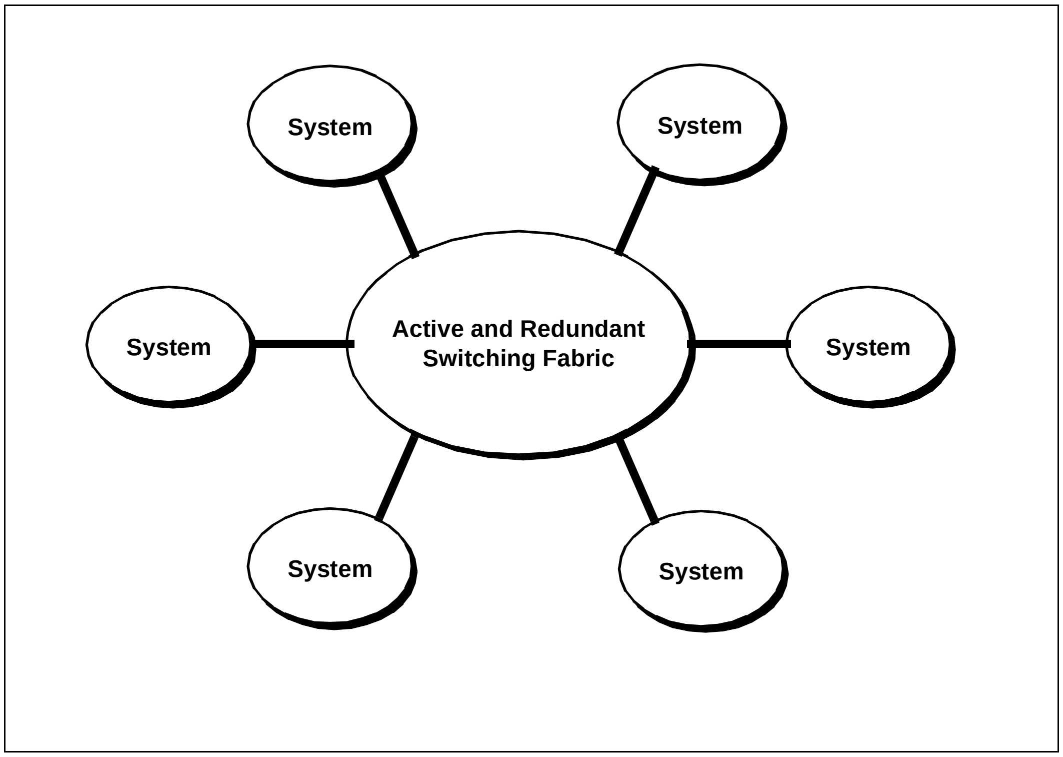
A system within a cluster may simply be built of standard non-redundant components, or it may have some internal redundancy to improve its resilience — for example, mirrored discs, redundant power supply and fans.

The system components can also improve the systems integrity by checking their internal operation for correctness – for example parity on data paths, and ECC on memory. This kind of checking improves the speed that faults can be detected and helps contain the damage caused by otherwise silent faults.

4.3.1 System Components

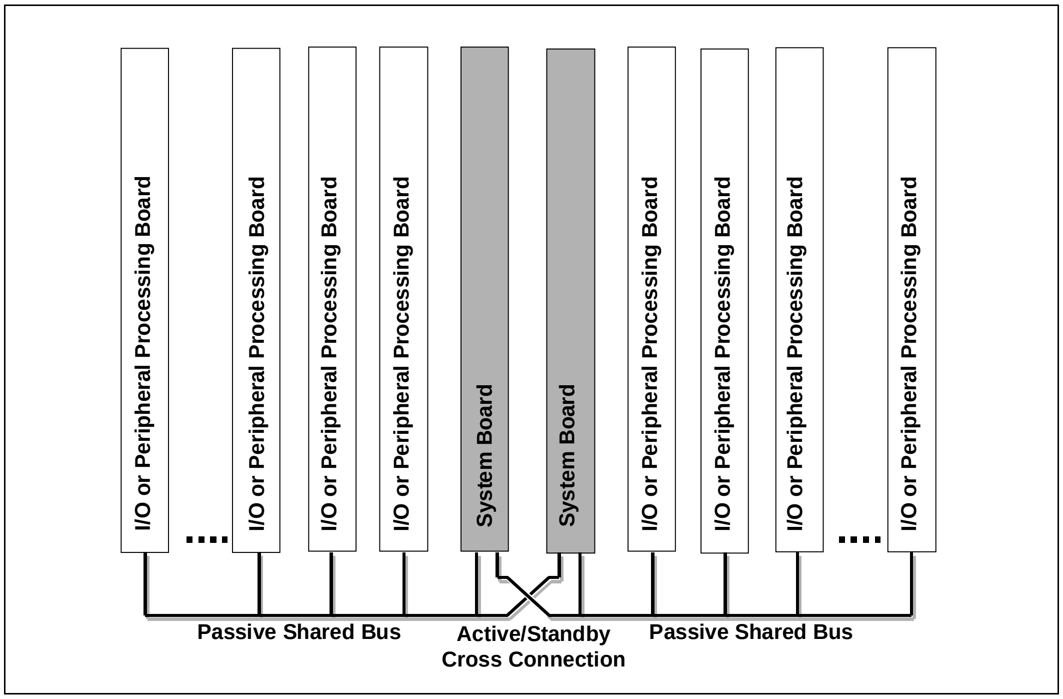
An HA cluster is composed of several systems, and each system has many individual components. Each of these components has a specific role in the application – and each may also have different roles, effects, and participation in the overall system availability, and in an open HA framework.

Figure 9. Clustered System Arrangement



In addition to a cluster environment, HA arrangements can be implemented within a systems by providing redundant internal components and data paths. For example a typical redundant CompactPCI system might consist of dual system boards, dual PCI busses, and a redundant set of I/O cards or peripheral slot processing cards as shown in Figure 10.

Figure 10. Redundant CompactPCI Arrangement


I/O or Peripheral Processing Board
I/O or Peripheral Processing Board
I/O or Peripheral Processing Board
I/O or Peripheral Processing Board
System Board
System Board
I/O or Peripheral Processing Board
I/O or Peripheral Processing Board
I/O or Peripheral Processing Board
I/O or Peripheral Processing Board
Passive Shared Bus
Active/Standby Cross Connection
Passive Shared Bus

Each of the two system cards has two CompactPCI busses – usually each card drives only one CompactPCI bus (and half of the I/O cards, but in the event of a system card failure, the remaining system card can manage both busses and all the I/O cards).

Each individual component that participates in the open HA framework and is used within an HA cluster must be represented by a standard model defined by the Open HA framework. When the Open HA framework is first introduced there must be a minimum set of qualified components.

A minimum set of components might include:

- Processor modules and memory (e.g., IA32 and IA64)
- Internal and external disk storage and storage I/O (e.g., IDE, SCSI, FC, and RAID)
- Network I/O cards (e.g., Ethernet, T1/E1, and ATM)
- Removable media (e.g., CD/DVD, DAT tape, and floppy disk)
- Chassis components (e.g., power supplies, fans, and backplanes)
- External fabric switches (e.g., Ethernet, ATM, and InfiniBand)
- Operating systems (e.g., Linux*, Microsoft Windows*, and Unix variants)
- HA middleware (e.g., configuration and fault management)
- Persistent data handling (e.g., HA Database, HA file system, and HA data replication)
- Protocol stacks (TCP/IP, SS7, OSI, etc.).
- Application subsystems and processes

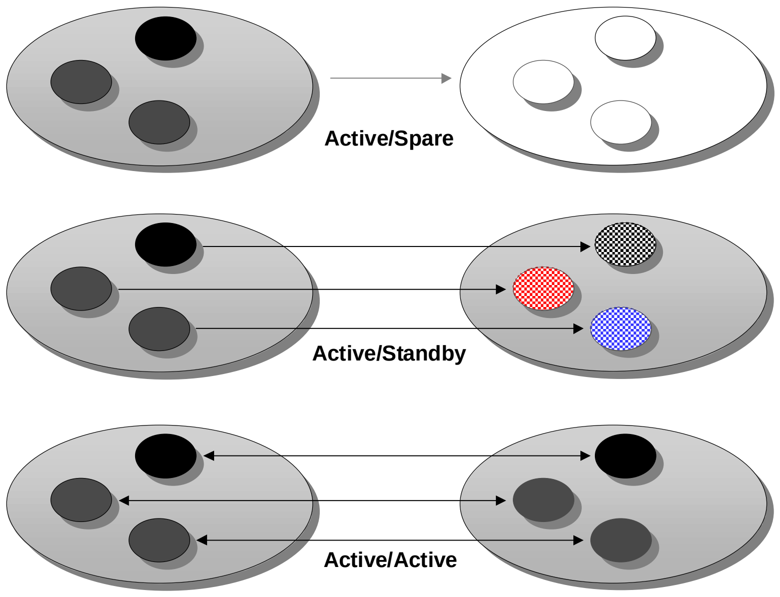
There must also be a set of qualification criteria and procedures for adding new components to the HA framework.

4.3.2 Application Environment

The HA framework defines an operating environment for the application. Different applications will have different needs and expectations of the operating environment, which include:

- Non-HA aware (active/spare) — these applications are written without any special coding to take advantage of the HA framework features. These applications must be cold restarted after a failure has occurred.
- Active/Standby mode — these applications need to have some knowledge of the HA state of the system. They may do some sort of data replication and checkpointing to improve the recovery time. The standby application expects notification from the HA middleware when it must change state (e.g., standby to active). Other HA events may also be used for communication path changes, storage changes, etc.
- Active/Active mode — these applications always handle live traffic, but change the extent of their active domain depending on the state of the system. The HA middleware must notify the application of the relevant cluster events, so that it can take the necessary internal reconfiguration steps.

Figure 11. Application HA Models



The HA framework should provide a set of APIs for the HA-aware applications to interact with the HA middleware. These APIs include:

- HA middleware registration
- HA event notification
- Notification of HA middleware of application state changes
- Data replication (control and data)

4.4 Availability Requirements

A system's availability requirements are normally stated in terms of the availability of the service it provides to the users of the system. Since many of the target applications for the Open HA Framework are the systems required for basic communication infrastructure services, many of these requirements are quite demanding. It is simply not adequate to have the service down while a system is repaired. Nor is it acceptable to have the service down while planned maintenance or upgrades are performed a system.

In many cases market forces set the level of acceptable availability. However, in some regulated areas such as the telephone network, there are some legislated standards imposed with consequential penalties for failure to meet the specified service levels. For example a widely adopted standard for the telephone network has been ***5-nines***, or ***service availability*** 99.999% percent of the time. This means that the system can only be down for a maximum of about 5 minutes a year.

Generally, if a complete system is composed of a number of critical subsystems, then each of the subsystems must actually achieve a much higher level of availability to meet the complete system availability. Similarly if a subsystem can be downed by a number of different causes, then the total amount of downtime must be spread between these different causes. These causes include:

- Hardware failure
- Operating system failure
- Application failure
- Application load and congestion
- Operator error
- Environmental problems (power failure, fire, earthquake, etc.)
- Planned downtime for system upgrade or change

4.4.1 Recovery Times

The allocation of available downtime between these kinds of causes is clearly an application and implementation specific task. However some of the consequences of this allocation usually are:

- The allocated time for hardware and operating system failure is generally a small portion of the 5 minutes per year, and is often 1 minute or less in a 5-nines system.
- The total recovery time (from failure, detection, reconfiguration, and restart) available for each failure depends of how often a failure occurs in a year, but can be as low as 1 to 5 seconds. The frequency of outages is often recorded and used as a measure of system un-availability.
- The HA design of the systems components must consider (and usually provide for) a rolling change and upgrade strategy.

Choosing smaller failure group sizes (to confine the failures to parts of the total system) can reduce total system downtime from individual failures. (This assumes the application can be partitioned, and total downtime can be pro-rated over the whole system.)

4.4.2 Repair and Testing

The speed and accuracy of repairs can have a significant impact on availability. Ideally when a hardware component fails, the systems can accurately identify the field replaceable unit (FRU) the first time. Then, once a replacement unit has been installed, it can be fully tested before being put into service for use by the system.

Typical requirements are:

- A failed FRU must be identified correctly by the system so that it can be replaced correctly on the first attempt 95% of the time.
- Failed FRU identification and replacement must not affect the service availability.
- A replacement FRU cannot be put in service until fully tested within the system.
- Testing a replacement FRU must not affect service availability.

The frequency of failures and the urgency of repair also affect the cost of operating a system. The more often service personnel have to visit a system to make repairs, the higher the operating cost. FRU reliability can be increased by choice of parts, as well as by providing redundancy built into the FRU.

4.4.3 Upgrades and Changes

All systems inevitably need changes to add or remove hardware components, install operating system changes, and install new application versions. Having the service down during such upgrades will usually violate the availability requirements, so some sort of rolling upgrade strategy is needed.

The rolling upgrade often requires that parts of the system are shutdown and upgraded while the redundant components keep the service available. Once the first parts of the system are upgraded, they are put into service, and the remaining parts are in turn shut down, upgraded, and returned to service.

The implications to the Open HA Framework and application are that some level of compatibility between versions of software is needed. These may include:

- Message formats, files, and data, common between different versions of software is compatible, or is tagged with version numbers for specific handling.
- Behavior of the different versions of software is compatible.
- Testing of the new software on the split system is possible before bringing it into service.
- A roll-back plan is in place in case the new software does not work.

When hardware changes are made, the operating system and management middleware must be able to assimilate the new configuration into the operating environment. New redundancy rules may be needed when components are introduced or removed from a system, and testing of new hardware is necessary before putting it into service.

4.5 HA Configuration and Cluster Management

Configuration and cluster management middleware is the key controlling entity in an HA configuration and is implemented by HA middleware distributed amongst the HA cluster systems. It maintains a system model of the components that comprise the cluster, defines how faults are detected in the cluster and what action should be taken as a result. When failures occur it takes the appropriate actions to reconfigure the cluster and notify and/or restart affected parts of the application.

The HA cluster configuration and management must be able to operate in a heterogeneous cluster. The member systems in the cluster may be of different types with different operating environments, and may have HA management middleware from different suppliers. The system model, the message protocols between cluster members, and application APIs must allow proper heterogeneous interoperability.

4.6 Data Replication and Data Integrity

One of the most difficult aspects of HA application design can be the preservation of the dynamic data and context despite component failures. Not only does it affect the design and testing complexity, but it may also have significant impacts on the system’s performance, dependability and accuracy.

An application usually stores long term data and context in a file system or database, but often has other more transient data that represents the instantaneous context of dialogues and/or transactions. The ability to efficiently replicate this kind of data can dramatically improve the application recovery time.

The open HA framework needs to provide for at least some of the following:

- An HA file system native to the operating system environment
- An HA database system
- An HA shared memory subsystem
- An application context replication and recovery subsystem

The HA file system, database and shared memory need to be integrated with the overall HA configuration and management subsystem so that their operation can be coordinated with the HA reconfiguration activities (e.g., failure, repair and recovery handling).

Additionally the API to a file system or database may have additional or modified calls and returns in an HA version when compared to a non-HA version.

5.0 System Capabilities – Configuration Management

5.1 Introduction

Configuration management involves knowing what types of hardware, firmware, and software components are actually in a system. It also tracks the intended configuration of the system (the ***system model***), which may or may not match the actual system configuration. Finally, configuration management contains the capability to modify the configuration of the individual components that comprise a system.

In order to provide accurate fault management, one must know the intended make-up of the system. The actions taken by a fault management system typically require changes to the system configuration, and this requires corresponding capabilities in the configuration management strategy. This is attained by maintaining an inventory of the system's components (hardware, firmware, and software) and by accurately maintaining information on the health of each component.

Although the title of this section is "System Capabilities – Configuration Management", the capabilities discussed here are only the sub-set of configuration management activities that are necessary in support of high availability. Details of fault management are outlined in Section 6.0.

5.2 Characteristics of System Components

System components are the hardware, software and data that comprise the system. A component is any part of a system that is individually managed. In infrastructure equipment, components (managed or not) typically include:

- **Hardware** – Disk drives, boards, fans, switches, cables, chassis, lights, and power supplies
- **Software** – Operating systems, drivers, servers, content, and applications
- **Logical Entities** – Databases, network topologies, routes, clusters and geographical regions

To lay the foundation for the sections that follow, there are six characteristics of system components that should be addressed in modeling and controlling a system:

- **Heterogeneity**. System components include hardware (boards, drives, fans, LEDs, etc.) and software (O/S, drivers, applications, middleware, etc.)
- **Transience**. System components are available or not available, according to either their health or their presence in the system due to a service event or upgrade
- **Dynamics**. The state of system components is typically dynamic and includes attributes such as health, operational state, and administrative state
- **Controllability**. Certain system components can be controlled from other parts of the system or through external system interfaces
- **Dependency**. System components typically have dependencies upon one another. For example, an application depends on the O/S, which in turn depends on the hardware.
- **Redundancy**. Some critical system components are redundant and are represented as logical groups that provide a single service within the system. For example, a system may have redundant fans, CPU cards and power supplies so that the system can continue to operate in the event that an individual component fails.

5.3 Dynamic System Model

5.3.1 Introduction

The ***system model*** provides the basis for both configuration and fault management and is a critical component in meeting availability targets within an HA system. The model is typically implemented in an in-memory database within the management middleware. This complete system model may use information from models of other components, such as a model of the hardware. Information about these component models may be made available via OS hooks or hardware management processors. The discussions on data collection in section Section 5.4 cover this in more detail.

A system model is an abstract description of component capabilities, interfaces, relevant data structures, and interactions. The system model dynamically tracks all of the ***managed components*** in the system and incorporates the configuration, state, relationships and dependencies of these components.

5.3.2 Concepts

The information contained in the system model should contain:

- **Actual System Population** – The Actual System Population is a reflection of the components available in the system at any given point in time. This is a dynamic population, especially on systems with hot-swap and hot-plug capabilities. This is essentially a census of the system component population.
- **Intended System Population** – The Intended System Population is a reflection of the components that have been, are currently, or are expected to be part of the system. It should be possible to remove components from the intended population if the system has changed to the point where some of the components no longer could be used in that system.
- **Component Information** – Contains static information about the component including the class, product, manufacturer and revision of a component. For physical components such as an assembly, this information is also known as the field replaceable unit (FRU) information.
- **Component State** – Contains the dynamic information for individual system components that are relevant to availability management. For an HA system, the information should be dynamically updated as the actual state of the components changes. For example, the X.731 state model supports attributes such as administrative state, operational state, usage state(s), availability status and control status [X.731].
- **Component Role** – In an HA system with redundant components, these components have roles such as active, standby and spare/unassigned. ***Active*** means that the component is actively engaged in providing service; ***standby*** means that the component is idle and waiting to take over. ***Unassigned*** or ***spare*** applies to components that are neither active nor standby, but are available to be made either active or standby. When an active component fails, the standby component is promoted to take over the active role. The roles are maintained in the system model.
- **Physical Dependencies** – System components have physical parent/child dependencies that need to be understood in order to manage the availability of a service. Children depend on their parent's presence and health for them to be assigned work in the system, while the parents depend on the presence and health of their children for their own proper operation. For example, a system's external I/O may depend on an I/O card, which in turn depends on the system power. Knowledge of these dependencies enables a management system or an operator to determine how a fault in one system component affects other components within the system.

It also allows for the determination of how modifying (disabling, changing, adding, etc.) a component affects the other components within the system.

- **Logical Dependencies** – As discussed in Section 4.0, system components may also have dependencies beyond the traditional physical dependencies. For example, an application on a system CPU card may depend on a database located on a separate CPU card. These dependencies are often represented using a dual-connected digraph (directional graph) in which each component is linked to every other component upon which it depends, and each component is linked to every component that depends on it. The system model contains sufficient information to be able to understand and track these complex dependencies.
- **Service Groups of Redundant Components** – The system model should support the ability to represent groups of redundant components as logical services called service groups. A service group corresponds to the ISO term 'protected group.' The system model maintains the redundancy policy and the operational role (active, standby or spare) of each component according to the component's state and the policy of the redundancy group (i.e., N+1, 2N, N+M, etc.). An example of a service group is the power to the chassis. Multiple power supplies work together to ensure uninterrupted power. These supplies are grouped together for management purposes because they are interchangeable and provide the same service. While other components in the system may depend on power being available, they do not depend on any individual power supply being operational. Therefore, other system components depend on the power service group, which logically represents the individual power supply components.

5.3.3 Approach

Before system deployment, an architecture, which encompasses the appropriate fault domains and redundancy model, is first developed and is implemented as the system model. Active and standby components are configured per the requirements of the system model. Component dependency information is also placed into the system model.

When the system boots up, the actual system component population is created. System components are automatically detected and added to the population census. As the system operates, the system is continuously monitored. The actual system component population is updated to reflect any additions, changes, or deletions to the population of system components.

Dependency relationships between the system components are continuously monitored. Changes (additions, deletions, or modifications) to a component, which has a dependency relationship with another component, must consider the effects on the dependent component before executing the change operation.

The system model also provides methods to control the configuration of the system components. Within this capability is the ability to select the appropriate component to succeed as the primary when performing a switchover. This switchover capability is discussed in Section 6.0 as part of Fault Management.

5.3.4 Techniques

- **Determining Actual System Component Population (Auto-detection).** This technique determines what components are in the system. It allows for the isolation of system components that are not required for the service. Components added to system (hot-swap/software installation) can be automatically detected and identified before components are enabled. Automatic detection and identification occurs similarly for component removal.

- **Verifying Required System Component Population (System Model).** This technique determines what components should be present in the system to provide a particular service. It works from a defined system model, detecting which components in the system model are present, and if those components are functional. Interdependencies among the components are also tracked and analyzed.
- **Obtaining Detailed Information about System Components (FRU).** This technique allows the system to obtain specification information about each component. Commonly referred to as ***field replaceable unit*** (***FRU***) information, it includes such items as: class, product, manufacturer, and revision. FRU information of system components is necessary for the repair step in fault management. The capability to obtain FRU information can reduce the MTTR by allowing a service technician to correctly identify replacement components before physically inspecting the component. FRU information can also be used during hot-swap events to determine if the component is needed in the system and if the appropriate supporting components are available (a device coupled with a device driver). Hot-swapped components can be left isolated during the time in which additional components are discovered.
- **Establishing System Configuration.** In establishing the system configuration, the system components are assembled to establish fault domains, necessary to provide the appropriate level of redundancy. Once all of the system components have been detected and identified, system configuration can establish relationships between components to establish redundancy.
- **Data Collection.** Information is collected and consolidated within the system model in order to monitor the health and state information of each component in the system. This information is available both locally and remotely.
- **Role Assignment.** When redundant components are grouped together to provide a service, they are assigned various policies such as N+1 or 2N. To support a specific policy, components are assigned operational roles including active, standby and spare. In the event of a component failure, these roles may be reassigned to maintain service. For example, if an active component fails, the standby component may be reassigned the role of active.

5.4 Interfaces to System Components

5.4.1 Introduction

Health status and state information from each system component is communicated to other system components within a system. This is referred to as intra-system communication. This information may also be reported externally to the system. When reported externally, it may be reported in a raw form, indicating the health status and state of each component, or it may be summarized to indicate the aggregate health of the system as a whole.

System components have states and need to be managed and controlled in a coordinated fashion. While these components may vary widely in type and function, an HA system must be able to 1) access the components' ***attributes*** to obtain state and configuration information, and 2) control these components via ***methods*** for reconfiguration or fault management.

There are several service availability functions both within the system (i.e., management middleware) and external to the system (i.e., network management system) that need to access and control the various system components, including much of the fault management functions discussed in Section 6.0. These functions need to be able to:

- Manage faults
- Monitor performance

- View or modify configuration
- Monitor applications
- Enable a system administrator to remotely access and control the system and its components
- Enable a network management system to interface to the system and its components

The key interface areas include the platform interface and the application interface.

The platform interface should provide other parts of the system (i.e., the operating system, management software or applications) with chassis management functionality including power supplies, fans and LEDs. The platform interface should also support hot-swap to allow the insertion, detection, initialization, configuration and removal of system peripherals or cards.

The application interface should provide applications with the ability to allow detection, registration and monitoring by other parts of the system. It should also enable the applications to participate in the system's fault management mechanism.

5.4.2 Objective

To provide a means by which management middleware and other system components can access state and configuration attributes of the various components within the system, as well as control these components via operational or configuration methods.

These attributes and methods may be utilized both within the system and external to the system.

5.4.3 Concepts

Health Status Information. Health status information indicates the healthiness of a system component. It does not include fault information, which is covered in Section 6.0. Health status information, for example, could be based on CIM [CIM] or the ITU X.731 recommendation [X.731] as a model.

State Information. The state information allows for the tracking of the system component's role within the redundancy model (i.e., active, standby, spare). State changes should be reported as an autonomous message on one or more of the management interfaces.

Monitor Information. In general, system components with a monitoring capability should collect information continuously, even if it is not being polled. The component may also attempt to notify management interfaces if an exception event occurs.

Remote Monitoring. Health status and state information may be monitored by an external management system.

Local Monitoring. The system can monitor itself and all of its system components for health status and state.

Administrative Control. This operation allows a local or remote management entity to invoke administrative actions on managed system and layer objects.

Fault Injection. Fault injection is the activity by which a failure condition is forced upon a system component. It is used to verify that the fault detection, isolation, recovery and repair mechanisms are functioning properly.

Diagnostics. Diagnostics involves testing of the system components. This may be done while the system component is on-line or off-line. Diagnostic testing may also be done destructively or non-destructively. Testing that interrupts the normal functioning of the system must be coordinated so it is either done during off-peak hours or another redundant component can handle the normal system traffic during the testing period.

Autonomous. Health status and state changes are reported as they occur. Alarm information (see Section 5.5) is an example of this form of component communication.

Directed, Polled. To get information, the requesting Management Interface function directs queries for Health Status and State (status) Information to a particular component(s), and the addressed component responds. This is typically, a two-way communication. This action is generated on a periodic basis, with a periodicity that ensures that important information does not overflow the local storage capacities of the components (registers, data structures, etc.)

Directed, Adhoc. Same as the polled action, except generated as needed.

Directed, Control. This communication allows a Management Entity to issue a command to perform an action. This may be a one or two-way form of communication, and may use a reliable or unreliable delivery mechanism.

Multiple Managers. Health Status and State information may be important and subject to management from multiple management functions. For instance, it may be used by the middleware layer, may be accessed through the local trade interface, and reported (directly or through the middleware layer) to one or more tiers of a formal network element management schema.

Management Interfaces. These are defined interfaces to communicate component information (see Section 5.5).

5.4.4 Approach

During the operation, as the state or health status of a system component changes, these changes must be reflected in the System Model. There are two ways of reporting these changes internally to the system. The first way is via Asynchronous communications methods. The second way is via a synchronous, or polled communications.

Health Status and State Information may also need to be reported to a manager external to the system. Either individual system component information may be conveyed or an overall view of the system's Health Status and State may be conveyed.

It may also be desirable to check the status of a system while it is not operational. This capability allows a system to be diagnosed and discovered when it is not functional due to failure or due to it not yet being started up.

Monitoring Health Status

A system may provide trending information by continuously monitoring the health of each component in the system. Health Status monitoring looks at the healthiness of a component which has not (yet) incurred a fault. Once a fault occurs, the component is logged as entering a faulted condition, and the Fault Management techniques discussed in Section 6.0 are applied.

Even after a component has failed, there is a need to monitor (or view) the health of that component. The ***Fault Management*** (***FM***) service's capabilities take care of recovering from the failed component, but the FM service's link to the operator is through the ***Configuration***

Management (***CM***) services. The operator uses the services of the CM to view the health of each component, determine which component has failed, and identify the component (***field replaceable unit***, ***FRU***) so that it can be replaced. Once the component has been replaced, the CM service is involved in re-establishing the system model (redundancies, etc.).

Health status may be reported to the manager by the individual components (in the form of event notifications) or they may be ascertained via a query mechanism from the manager. The manager may also use testing and diagnostic techniques to determine the health status of an individual component (or a set of components). These techniques are discussed in Section 6.2 and Section 6.5.

As system components move from the healthy condition to the ***degraded condition*** – where, in the degraded condition, an abnormality has been detected, but the component is still delivering an acceptable service level — the system manager can anticipate a system failure in the future. Several options are available to the system manager at this point, depending on the severity of the degradation and the policies and procedures, which are in place. Some of these options are:

- Off-loading work from the degraded system component to a more healthy one
- Notifying a system operator
- Performing fault management recovery and repair measures (refer to Section 6.0)

Implicit in this set is the capability of system components to report (via event notifications or in response to a query) their health status. Also implicit in this set is the capability of a system manager to receive (via event notifications or queries) this information and process it accordingly.

Monitoring State Changes

Similar to monitoring heath status, it is also important that the CM service monitors changes that occur in the state of any component. The component states of active/standby as well as the current operational state need to be recorded. Depending on the application, the CM service may also keep track of what data object an application is processing, and how far through that processing the application is. This type of data may also (or alternatively) be stored by an application that is in the standby mode for the current application.

Testing and Diagnostics

The CM service has the responsibility, though the system model, of understanding which diagnostic tests can be run on a component. Additionally, it can determine which tests can be run while the system or component is being used by the system and which must be run only when the component is not being used. The CM service provides this information to applications and management middleware which can then run the diagnostics.

In many very high reliability systems, diagnostics are run on components that are not in service to ensure that they will be ready if needed. Additionally, some systems may run diagnostics periodically on components, while the are either in use or in standby, to check that they are working properly.

5.4.5 Techniques

Industry Standard Methods. SNMP, RMON, CIM, TMN(OSI CMIP), IPMI and CORBA are several of the industry standards available for reporting information external to the system. These methods may also be used for communicating information between the management function and the system components. A further discussion of these terms is in Section 5.5.3.

The Intelligent Platform Management Interface is an industry standard method that provides a means by which hardware-based system components may report health status information. This allows a system to be monitored even if it is not fully functional or operational. Implementing IPMI requires specialized hardware, typically including a separate microprocessor or microcontroller, which is used to gather, maintain, and report system status to a remote device.

Alternate Methods. Any method by which a system component may communicate with the system manager will provide a means by which Health Status Information may be conveyed. Message passing, queuing, semaphores, and shared memory are some examples of alternate methods of communication.

5.4.6 Dependencies

External communications are constrained by the requirements of the external management system.

All communications are constrained by the communications media contained in the system and the traffic bandwidth of that media.

5.5 Management Interfaces

5.5.1 Introduction

Implementations of this HA system model typically have multiple management interfaces that support different functions and facilitate different types of action.

Interprocess. At the most fundamental level there is a primitive management interface between processes. From a functional perspective, this information is ‘in-band’ and includes information like *error and return codes*. This interface communicates status and control information and should use open-standard API and methods, when available. This interface usually supports very terse context and information content.

Interlayer. In the system model there is a need to communicate Health, Status and State information from one layer to another, for instance from the hardware to the OS layer. The Management Middleware, which provides immediate configuration and fault management of the system, is a specific form of this interface. Depending upon the need and method of request, the context and information content can vary from terse to extremely detailed information content.

External Management. The administration of the system typically involves layers of different management functions, context views, requirements and management objectives. These management interfaces could include one or more of the following types:

- **Network or Element.** This is typically a structured remote network management schema that defines the management interfaces, communications, elements and Management entities (which are often multi-tiered).
- **Local Administrative.** Most network elements have some form of local administrative management interface. They are used to provide information and may provide control access to system components. Such an interface can be as simply as blink (or beep) sequence, equipment-specific LED’s, local alarms, remote alarms (email, pager, landline call), system console, or other trade interface (TL1, or local browser support).

5.5.2 Alarms

Alarms are intended to convey critical system exception information in an appropriate, effective and timely method.

Definition. Alarms are typically autonomously generated messages that are triggered by a specific causative stimuli. The stimuli might be: a ***detected fault***; a ***state change*** (system power-on generating a trap, or role change generating an alarm); a ***threshold crossing*** (a specified parameter, or rate-of change of that parameter, exceeds or falls below a pre-established value); or when a particular ***event type*** or ***event severity*** is recorded (for instance when the storage medium is running out of available space). A filter process is often used to determine alarm events and to determine the level.

Concept of Levels. A management scheme may include different escalating levels of alarms. These may be different LED colors of different types (alarms, alerts, warnings, traps). The levels may be differentiated by severity or type and may be escalated if not resolved after some defined interval.

Context/Content. To be useful the alarm should include sufficient information on the context of the exception, the type and severity (if available) and specific information on the location.

Communication Method. The alarm information can be broadcast to numerous management interfaces and to any management entity. For instance, a power-on trap message might be published for everyone to hear, and might be published without regard to whether any entity received it (unreliable). Alarms can also be directed to particular management entities, and can be unreliable (as described above), periodic (generated at a interval until resolved), or can be persistent (continuing to alarm until the alarm is acknowledged, even if the fault has not been resolved). For instance, the loss of the co-generation facility at a central office would generate an alarm message that would be both persistent (ensuring that all appropriate management interfaces – local and remote received the message) and periodic (re-alarming at periodic intervals while continuing to operate on backup battery power).

5.5.3 Integration with External Network Management Systems

It is critical that a system not only be able to control and monitor itself, but also that it work in a network of other systems. There are several standard methods that external management systems use to communicate with the systems they monitor. Most of these methods have defined messages with room for expansion, and when they are expanded, there is usually an attempt to standardize messages that compatibility is optimized. The following standards are in use today:

Simple Network Management Protocol (*SNMP*). SNMP is a set of protocols that pass information about whether or not a component is operating properly. SNMP uses ***Management Information Bases*** (***MIB***s) to store data about a system.

Remote Monitoring (*RMON*). RMON allows network usage information to be gathered by providing new sets of MIBs. Network devices in a system must be RMON compliant or RMON will not work.

Common Management Information Protocol (*CMIP*). CMIP is an OSI standard protocol used with CMIS (common management information services). CMIS defines a system of network management information services. CMIP was proposed as a replacement for the less sophisticated SNMP, but has taken root only in the telecom space. CMIP provides improved security and better reporting of unusual network conditions.

Web-Based Enterprise Management (*WBEM*). WBEM is a standard being driven by the DMTF (Distributed Management Task Force) for managing groups of computers connected in a network. There are several standards feeding into WBEM, including Common Information Model (CIM) and Intelligent Platform Management Interface (IPMI).

5.5.4 User Interface

A user can see what is going on within a system from three positions:

At the system. Indicators to show which components are active, standby and failed are a requirement of maintainability. If the indicators are not local to the component that fails, it is too easy to pull out and replace the working component instead of the failed component. Additionally, local alarm indicators are useful in case the remote management console fails to detect an error.

At a local console. A GUI screen is useful for debug and configuration, however in most large applications, no one will ever monitor the system state from a local console.

At remote consoles. Most of the techniques specified in Section 5.5.3 are designed to enable monitoring of a set of systems from multiple locations over a network. The locations may be local to the building, across the country, or both.

The function of monitoring a system is critical in high availability systems, so monitor and control functions must be available from multiple sites. When the system is designed, the methods and locations of monitoring must be reviewed to ensure that the three monitoring positions are covered.

5.6 In-Service Upgrading

5.6.1 Introduction

System administrators need the ability to perform hardware and software upgrades without interrupting the service. To support this, the system should provide redundant components so that operation can be offloaded from the component that needs to be upgraded. In addition, the system should provide support for ***hot-swap*** so that hardware components can be inserted into or removed from the system while it is operating. Similarly, the system should enable software components to be upgraded without impacting service. This is known as a ***rolling upgrade.***

5.6.2 Objective

Enable on-line upgrades of hardware and software without interrupting the service.

5.6.3 Concepts

Hot-Swap. The ability to insert or remove a hardware component while the system is powered and operational.

Rollback. The ability to regress to a previously known good version if the upgrade is unsuccessful.

Split-Mode Upgrade. For systems with redundant components, split-mode upgrades enable the upgrade of one component at a time by transferring system operation to the component that isn't being upgraded.

5.6.4 Approach

Systems are comprised of a broad range of components that need to be upgraded, including hardware, operating systems, applications and peripherals. In systems designed for service availability, many of these components are redundant. This enables the system to transfer operation away from the component that needs to be upgraded without any outage to the service.

5.6.5 Techniques

Hardware upgrades require the ability for components to be removed from and inserted into the system without requiring a reboot or the system to be powered down. This capability is called hot-swap and is a major feature of new platforms such as CompactPCI. Hot-Swap requires hardware and operating system support to enable the capability.

Operating systems and other software components can either be dynamically upgraded during operation, or the system can transfer operation to a standby component to enable the upgrade to occur off-line.

5.6.6 Dependencies

In-service upgrading depends on essentially the same functions as repair. All components that need to be upgraded while the system is service must have the ability to be hot-swapped in and out of the system. In the case of hardware, the software associated with that hardware must also be capable of being dynamically loaded and unloaded.

6.0 System Capabilities — Fault Management

Managing faults in a system is typically a five-stage process.

1. ***Detection*** – The fault is found
2. ***Diagnosis*** – The cause of the fault is determined
3. ***Isolation*** – The rest of the system is protected from the fault
4. ***Recovery*** – The system is adjusted or re-started so it functions properly
5. ***Repair*** - A faulty system component is replaced

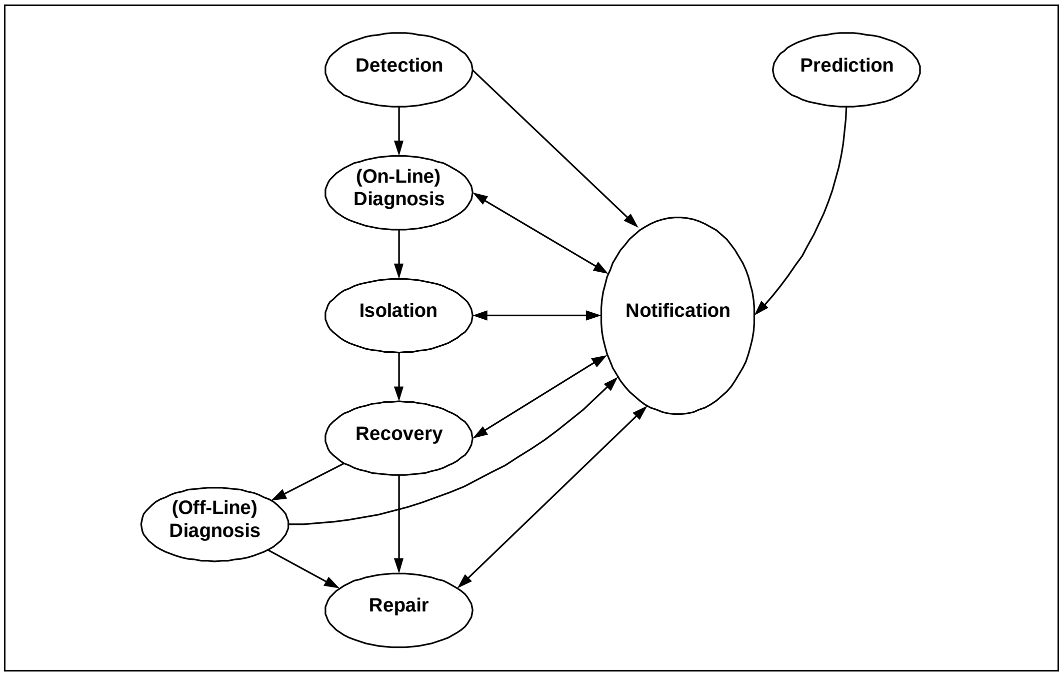
Notification of the fault occurs at many points in this process. Between each step there is notification of the fault to the next step or steps in the process. On fault detection, notification may occur during the diagnosis, isolation and perhaps recovery software components simultaneously. There will also be notifications to the system model and system operator indicating the status of the components. A general discussion on notification appears at the end of this section.

It is important to note that there are fine lines between some of the above stages. For the purpose of avoiding overlap in discussion, this document will use the most restrictive definition for each stage:

- **Detection.** A fault is found, but determination of the failed component is not made
- **Diagnosis.** The determination of which component has failed
- **Isolation.** Ensuring a fault does not cause a system failure (isolation does not necessarily make the system function correctly)
- **Recovery.** Restoring system to expected behavior
- **Repair.** Restoring a system to full capability including all redundancy

A final part of fault management is fault prediction. Fault prediction is an alternate form of fault detection, which includes built-in diagnosis. Based on predicted faults, the system operator can be given the opportunity to preemptively perform an on-line repair rather than wait for a fault to occur.

Figure 12. Fault Management Flow Chart



A fault occurs when a system component is not performing as expected. The severity of the fault can be evaluated by its effect on the service availability level of the system as a whole. If a backup component is available and is able to assume at least some of the failed component's responsibilities, a level of service availability is maintained. If the faulted component is required to provide service and has no backup or load sharing capabilities, then a service interruption occurs.

The rate at which faults can be detected directly affects the time it takes for a system to recover to its full capabilities. System designs that invest heavily in component reliability and/or fault detection capabilities may require less investment in fault recovery capabilities. In the end, all facets of the development and employment of a fault management system must be weighed against each other to provide the best solution for the required level of service availability.

6.1 Detection

6.1.1 Introduction

Detection is the process of identifying an undesirable condition (fault or symptom) that may lead to the loss of service from the system or device. Fault detection may be by direct observation – correlating multiple events in location or time, or by inference – by observing other behavior of the system.

Predictable levels of service availability can only be obtained through a methodical means of fault detection. Systems, which are intended for very high levels of availability, must include highly-responsive capabilities of fault detection to ensure that events that may lead to faults, events that indicate developing faults, active faults, and latent faults are quickly captured. In all but rare cases, fault detection capabilities are a required precursor to fault recovery.

6.1.2 Objective

The objective of fault detection is to detect when a fault occurs, and pass information on the fault to the components responsible for diagnosis, isolation and recovery. This information would include the location and type of fault, time of occurrence, and perhaps the most likely next affected component. For example, if a fault occurs in a multiplication subroutine, it is useful to also know which routine is expecting the result.

6.1.3 Concepts

Active Faults. Active faults are faults that have been detected, but have yet to be isolated and/or recovered from. Active faults may or may not degrade service availability. A system can maintain a level of service availability if the fault resides in a component that is inactive, has a backup, or has load-balancing capability. In this case the detected fault may remain active while the system is still delivering an acceptable level of service. An example of an active fault is when an active host system master board crashes. If this CPU board is contained in a system that has a redundant backup system master, then the backup host may detect that the redundant host is not functioning, at which point isolation and recovery activities may begin. As shown in the previous example, once an active fault is detected the fault management state will transition so that the faulty component can be isolated in an attempt to preserve service availability.

Latent Faults. A latent fault is a fault that eludes the detection schema and remains undetected for a period of time. Sources of latency include inactive components within the system, and uncovered fault scenarios. A latent fault, for example, may occur in a N+1 configuration with limited fault detection if the backup component fails. In this case the failed standby component will not be detected until another component failure occurs and the backup component is brought online. It should be noted that a latent fault is not impacting system serviceability when the faulty component or subsystem is not being exercised.

Fault Detector. A fault detector is a hardware or software component that checks for faults. A fault detector is ***triggered*** when that detector recognizes a fault. The term "***audit"*** is also used to refer to this type of component.

Direct and Indirect Detection. In direct detection, a fault detector finds an error in the output of a component. The component with the erroneous output is typically faulted. Indirect detection looks for more system-centric errors, such as high temperature or excessive memory use or CPU time. In this case, it is only known that a problem exists. The cause of the problem may be one or more components and diagnosis of the problem is needed before the cause of the problem can be determined.

Detection Frequency. Faults can be detected either synchronously or asynchronously. As the frequency of synchronous detection increases, the system increasingly exhibits degraded performance.

6.1.4 Approach

Fault detection, in its most basic form, is simply the ability to detect that abnormal conditions exist within the system. Detection may be by direct observation – correlating multiple events in location or time, or by inference – by observing other behavior of the system. Events in some circumstances may lead to the determination of a fault, while in other circumstances, it may be treated as normal system behavior.

Detection of faults can occur through various avenues within a system. A fault may be detected at the source of the fault itself. There are a variety of components, which are designed so that the component can trap or report error or out-of-tolerance conditions. These types of detected faults can range anywhere from slight threshold incursions to complete component or resource failures. An example of a fault, which is detected within a resource, is an intelligent power supply. It is common practice to design power supplies with various failure states (i.e., degrade, inhibit, failure, etc.). Based on the failure state reported, the severity of the fault can be determined along with the fault's impact upon system serviceability.

Faults may also be detected by system resources external to the faulted subsystem. One case of this might be a fan failure. A non-intelligent fan itself will not indicate that it is no longer working, but temperature sensor readings would indicate there is a fault in the cooling subsystem that needs attention.

In a system constructed to provide a high level of service availability, it is desirable that each component or sub-system within the system contribute to the fault detection process. This contribution can take the form of simply reporting current status of the failed or failing fault domain. In a more complexly-managed system the level or type of responses from various components within the system may be used to detect a single fault. The level of complexity of the fault detection capabilities of the system has a strong impact upon service availability.

6.1.5 Techniques

There are many ways of detecting faults. The following are some general methods that can be applied throughout a high availability system. A system does not have to employ all of these methods to be considered highly available, nor is this an all-inclusive list of fault detection methods.

There is a balance between detection and performance that must be established for a given system in a given application. It is possible to do so much checking that the main tasks progress at an unacceptable rate. It is also possible to reduce checking to the point that faults are not detected. The worst case is to have excessive fault detection on one data path, but little to none on another. Another point to consider when including large numbers of fault detectors in a design is that the fault detectors themselves can fail. For these reasons it is critical that fault detection be designed as an integral part of the system, by those who fully understand the system.

Value Range Checking

In most applications the result of an operation must fall within a certain range. Tests can be done for these boundary conditions to verify that the data is as expected. This concept is applied when setting limits for temperature or airflow and when doing address calculations and operations, particularly when it comes to stack and I/O operations.

In some cases it may be necessary to pass an expected output value range in addition to the data normally passed between components for this type of testing to be possible.

Data Integrity Checking

Whenever data is transferred from one component to another it is possible to get corruption. This is particularly true when data is passed between hardware components. However, since software layers can hide the difference between local memory transfers and transfers across remote links, it may be useful to check integrity at multiple points.

Data integrity can be verified using many methods, most of which depend on either redundancy or summary information included within the data. Some of the methods may use sufficient redundancy to not only detect an error, but also to correct it. However, most methods contain only enough additional information to detect that the data is not valid. Examples of typical methods include parity, checksums, and Cyclic Redundancy Checks (CRCs).

Comparison Testing

When redundant systems are employed it is possible to have two systems make calculations in parallel. The results are then compared, and a fault is detected if the results do not match. This concept is also called voting, and is discussed in Section 3.5. Comparisons can be made at any level of the system, from cycle-by-cycle comparisons on a memory bus to final output being sent over the network.

Time Testing

Time tests can be the simplest form of error detection. If an event is expected within a certain time frame and the event does not occur, a fault is detected. This concept can be applied in hardware, using watchdog timers, and in software, using either hardware timers or software processes.

One specific method of time testing is commonly referred to as ***heartbeating***. This technique, which can be implemented in both hardware and software, uses some type of message handshaking that is performed at a predefined periodic frequency. This technique is used to verify that the appropriate components or subsystems still maintain some level of functionality.

When redundant systems are used, time checks can verify that the systems are operating at the same rate, which would indicate that no faults are present. If redundant systems are not used, expected times can be set based on design parameters.

While time testing is useful for fault detection, it does not always give a good indication of where a fault occurred. For example, if a routine does not return in the expected amount of time, it is not possible to know if the routine failed or if the processor had unexpected interrupts.

Time testing works well in ***deterministic systems***, but can be problematic in systems where the times of events may not be completely deterministic. In this case, the non-deterministic performance of a system may cause time testing to sense faults that don’t really exist. On the other hand, time testing can catch non-determinism in a system that was assumed to be deterministic.

User and Other Observable Detection

There are some cases where the end user of the system will detect a problem that has not been detected by the systems management functions. It must be possible for diagnostics and other tests to be started by external command in order to resolve these issues. Ideally, as experience with faults detected in this manner grows, the automatic fault detection of the system can be expanded to find these problems.

Fail with Notification

This is also known as ‘don’t fail silently’. It is critical in HA systems that any fault, no matter how small or how infrequent, is logged or recorded in some way. This allows prediction of future failures and assures that if the system model indicates that all is well, all is indeed well.

6.1.6 Dependencies

Fault detection is heavily dependent on facilities designed into the system infrastructure. If a system is not designed to provide additional information or redundancy for detection of faults, many faults may go undetected.

6.2 Diagnosis

There are two sets of operations in HA systems that use the term diagnosis. The first are the immediate acts taken after a fault is found to isolate the fault and recover from it. The second set of operations are those used as part of the debug and repair process. The following section discusses only the first set of operations. The second set is covered in Section 6.5.

6.2.1 Introduction

Once a fault is detected, the problem must be diagnosed to determine the proper isolation and recovery actions. Diagnosis analyzes one or more events and system parameters to determine the nature and location of a fault. This step can be automatic or invoked separately as a user diagnostic. The diagnosis may be automatically acted upon or reported to an operator.The mechanism used to diagnose a problem depends on the type of system component (i.e., hardware, operating system, peripheral, application, etc.) being diagnosed as well as the system component responsible for performing the diagnosis. For example, a component such as an intelligent I/O card may be able to self-diagnose its own problems, while a fan may be diagnosed elsewhere within the system.

In some systems, a single fault may lead to multiple errors being detected. Identifying the originating fault (root cause analysis) is also part of diagnosis. This function is typically done after the faults have been isolated and recovery has occurred. Further discussion on this is in Section 6.5.

6.2.2 Objective

The primary objective of diagnosis is to determine the location of a fault so that it is possible to isolate the fault and recover from it. Section 3.4 and Figure 3 show that a fault in one component could be caused by faults in other components.

On-line vs. Off-line Diagnosis. Faults can be either diagnosed while the system is application ready (on-line), or when the system is not available for running applications (off-line). On-line systems can run off-line diagnostics on a given device by restricting the target device from being available to applications. Off-line diagnostics can refer to pre and post boot diagnostics that consume the availability of the system, thus reducing overall service availability. It is therefore possible to run some limited off-line diagnostics while the system is on line, but at reduced availability. However, many off-line diagnostics have availability costs that are only justifiable for debug, repair, or qualification functions, which are covered in Section 6.5.

Single vs. Multiple Failure Modes. A system resource may have a single failure mode or it may have multiple failure modes. When a resource has a single failure mode, the diagnosis is implicit in the detection event. When a resource has multiple failure modes, diagnosis is required to identify which failure occurred.

Local vs. Global. From the standpoint of the system, a diagnosis may be performed locally within a system resource, or it may be performed globally by another resource within the system.

Granularity of Diagnosis. The objectives for diagnostics in a particular system determine the granularity. From the perspective of service availability in a system with redundant components, the diagnosis at a minimum must be able to identify which system component failed. More granularity may be required to support the recovery and notification actions. There is a trade-off between granularity and performance, as diagnosis to a very fine granularity can use a large amount of processing power.

6.2.3 Approach

Systems are comprised of a broad range of resources that can fail, including hardware, operating systems, applications and peripherals. In systems designed for service availability, many of these resources are redundant. To enable rapid response in the event of failure, the first priority of diagnosis is to identify which component failed and needs to be isolated and repaired or replaced. The detection of a fault in a component that has redundancy, even if it has not yet been diagnosed to a granular level, is in many cases enough of a reason to switch operation over to the redundant component.

Faults can be diagnosed in several areas of the system. A component may have the ability to perform local, self-diagnosis. A failed component may also be diagnosed by a peer, another component, the operating system or the management middleware. Regardless of where the diagnosis is performed, it needs to identify the failed component and provide the results of the diagnosis to the decision-making entity within the system to allow proper recovery.

If the detection is local and there is only one failure mode, the diagnosis is implicit in the detection event. If there are multiple failure modes, additional work is required to further query the resource or to evaluate information that has already been collected for that component.

6.2.4 Techniques

Diagnostic procedures can be run by the component itself, a peer to the component, or by another program that is focused on diagnostics. Some diagnostic procedures may need OS kernel privileges, so they may have to be implemented at least partially within the OS.

Implicit Diagnosis and Self-Diagnosis

Implicit diagnosis is the simplest form of diagnosis. It is used when a component reports itself as failing, or a fault detector that looks for single modes of failure is activated. The fault directly implies the component that has failed or is out of specification.

Failures such as low fan speed or low disk-drive speed are typically done with implicit diagnosis, as only one component could cause that failure. Using indirect detectors usually make implicit diagnosis impossible. For example, high CPU temperature could be caused by low fan speed, blocked air vents, or a physical failure of the CPU or its heatsink. In this case further diagnosis must be done to determine the cause, and therefore the isolation and recovery process.

Self-diagnosis within a component is a subset of implicit diagnosis. When a component can find faults within its own operation, and then report those faults to the system, it is implicit that the component reporting the failure is the one that is failing

On-Line Diagnosis

On-line diagnosis is done while the system is running its normal tasks. This implies that the fault which created the need for diagnosis was not fatal to the system, nor did it require that a redundant component take over for the faulted component. Once the component suspected of having a fault is removed from normal system operation, further diagnosis is considered off-line.

For environment-centric faults, such as temperature and voltage, it is possible to adjust various system parameters and loading to pinpoint the faulty component. This can be done while the system is performing its normal functions.

Intermittent faults in the system are typically best diagnosed on-line, as it maintains service availability. Taking an entire system off-line to diagnose one component can reduce the availability of a system as multiple redundant components would be tied up with diagnostics and unavailable as standby components.

Examples of On-Line Diagnostics are:

- **Network and Message Faults.** These faults can be diagnosed by sending extra test messages. In many cases the fault can be diagnosed without saturating the communications channel.
- **Memory Faults.** These faults can be diagnosed by moving programs in memory to allow exhaustive tests on a block by block basis.

It is also possible to continuously run a set of on-line diagnostics. Doing this can be considered a form of fault detection, followed by implicit diagnosis. Running these procedures during normal operation reduces system performance, so the system architect must determine the appropriate trade-off.

6.2.5 Dependencies

Proper diagnosis depends on the following items:

- The ability of the system and/or its components to provide accurate detection of faults
- Having information that shows dependencies between components (an understanding of which components could cause a fault in which others)
- Having information about the fault detectors available to catch faults in components that are part of the dependency tree for a faulted component
- Having diagnostic routines which are effective (best done by those with intimate knowledge of the component)

6.3 Fault Isolation

6.3.1 Introduction

The process of isolation takes the defective component of the system out of service. The region that is isolated must be bounded at a point where it can be removed from all interactions with the system. The isolation is intended to insulate the system from the fault so it does not cause secondary failures. By isolating a fault in a running system, the system can be maintained in an operating state. Fault isolation can be accomplished in both physical boundaries as well as logical boundaries.

As noted in Section 6.0, there is a fine line between isolation and recovery. For this section, fault isolation includes actions that prevent a fault from propagating, but do NOT make the system function correctly. Actions that change a system from either an inoperative or degraded state to full operation are considered fault recovery and are covered in Section 6.4. The recovery and fault isolation steps may be combined inseparably. For example, recovery by redirection of IP addresses is both an isolation and recovery action.

A system can maintain a level of service availability if the fault resides in a component that is inactive, or is a backup and does not interact with the active components. In this case the detected failure may remain active while the system is still delivering an acceptable level of service. Since this latent fault is not impacting system service, the isolation step is considered complete in this case.

6.3.2 Objective

The objective of fault isolation is to keep a fault from propagating to other components of the system. This is done by removal of the device. If hardware is capable of actions like power removal, this action is also performed. The removal also includes proper interactions with the software components involved with the hardware that is powered off.

6.3.3 Concepts

Physical Isolation. To perform isolation at the physical level, the system must provide mechanisms to prevent the component from interacting in the system. This will require hardware mechanisms such as that provided in the slot control mechanisms in the PICMG 2.1 specification. This isolation consists of disconnection from the bus and powering off the module.

Data Isolation. In a software case, physical isolation can be accomplished with ***Memory Management Unit*** (***MMU***) support to prevent read, write or both on a page or memory region.

Logical Isolation. Logical isolation of a component in a hardware sense would be to remove the device entries from the I/O subsystem so that no further interactions with the hardware are possible. This is referred to as ***fencing***. This can also include interactions with the device driver to prevent interactions with the device or even removal of the device driver. Isolation in a software sense consists of removal of the component consistent with the system’s ability. For example, the following techniques can be used to remove a component: killing a process or task and removing it from the process table, unloading a loadable library, or removing offending files in a file system by renaming them or moving them to a non-used area.

6.3.4 Approach

Multiple techniques can be used for isolation. For a simple board/driver combination, the device can be turned off and the driver removed. Then further interactions with that component will fail but the system operation will continue. Components can be removed at various levels or granularity and hierarchy, from complete applications and drivers down to just instructing a driver to isolate itself.

Pushing the action of isolation lower can put it closer to the detection and thus be more responsive. The tradeoff is found in complexity. In general, quick action is better to prevent propagation. Localized and fast acting isolation methods must be weighed against a global impact to best maintain service availability. For example, if a power supply was indicating that it was below a low voltage threshold and there was more than one power supply available, then a quick reaction would be to simply isolate that power module to prevent it from impacting the system. A more global impact could be that the sum of the power modules were being asked to provide beyond their maximum capacity. Removing the first power module that indicated it was not able to keep up will then cause the remaining power modules to be even more overloaded, resulting in the reverse reaction where power modules would shutdown causing the entire system to fail.

In a system constructed to provide a high level of service availability to adequately perform isolation, a clear understanding of cause and effect of a component's existence in a system is required.

A software example of isolation would be that of a continuous spurious interrupt. If no component is indicating the need for service, and the interrupts continue to be fielded, then that could cause a disruption of service. An interrupt masking may be appropriate to isolate the problem. Again, it is important to understand the ramifications that masking that particular type of interrupt may cause on the system.

6.3.5 Techniques

Component Isolation

If a component is causing a detrimental effect on a bus or communication channel, that device can be isolated physically (using hardware) or logically (by requesting the system to stop communicating with it.) If ***logical isolation*** is not successful, ***physical isolation*** may be needed.

Quiescing Components

Even if a component is operating in a degraded state or causing a detrimental effect on a bus or communication channel, it may be desirable to have that component complete tasks it has in progress before isolating it. This can reduce the number of lost tasks which occur on switchover.

Quiescing is usually done by isolating the inputs, and then waiting for all output to stop before isolating the outputs.

Safe Value Output

If a component is supposed to output a value in a certain range and fails to do so, one could just block, or turn off that component. Unfortunately, this may cause an entire system failure. A second approach would be to have the component output a safe value that will allow the system to function (perhaps in a degraded manner) until a recovery can be implemented. ***Value coasting***, or maintaining the last valid value, is one method of providing a safe value output. Another method would be to use a pre-programmed value table.

It is important to note that if this method is used, the management system and all components using the output of this component must be notified of the failure. If this is not done it is possible that the failure would not be detected, and it would not be known that the system was using false data. This is critical in HA systems, as silent failures are not acceptable.

Routing Change

If a component provides a service that more than one device can provide, then another method of isolation could be to re-assign and/or remove the device from the service list. In the case of a networking device, this could be removing or reassigning the routing table to remove the failed component.

6.3.6 Dependencies

Fault isolation is dependent on the results of the diagnosis as well as the definition of the system dependency tree. The dependencies of software modules or hardware components are the active results of the mapping of the system (system model defined in Section 5.3) and the results of reliability modeling (system modeling for reliability in Section 3.3) The depth and thoroughness of the detection infrastructure is quite often proportional to the level fault awareness.

6.4 Recovery

6.4.1 Introduction

Recovery is the process of reassigning the necessary resources to restore the system to an operating state. Recovery also requires restoring any portions of the system that were adversely affected by the failing component. Recovery is the process of providing some level of service back to the systems. This can be in a reduced capacity if the isolated component does not have a redundant component or it could be the activation of a redundant component.

6.4.2 Objective

The objective of the recovery process is to restore the system to an operating state, even if it is in a reduced capacity.

6.4.3 Concepts

Rebalance/Re-route. In an ***N+1*** or ***N+M*** system, there is a hierarchy of component dependencies. When a component is determined to be bad, any other system components depending upon this resource must be recovered as well. This aspect of topology management becomes part of the critical path to restoring service.

Active/Standby. When redundant components are used, one is commonly in use (active) while the other is in standby, waiting to take over in case of a failure. The standby component is typically receiving either the same input as the active component, or is receiving information about what the active component is processing and what the last processed data is.

Checkpointing. The technique used to keep a standby component aware of where it should start processing when it takes over is referred to as checkpointing.

Reset/restart. A technique that can be used to recover a failed component is the process of resetting and/or restarting that component. The process would be performed on a standby component if the system has redundant components, or on the active component if it can be determined that it is a transient failure versus a hard failure.

6.4.4 Approach

In Section 3.3.3, the recovery action was briefly mentioned with respect to the ability to tolerate a fault. Several common techniques are used for the recovery action depending on the specific needs of the system and its application. For a fault management infrastructure to work, each method will need to be able to be implemented either directly or with the pieces provided.

Common techniques for recovery start with the ability to have some level of redundancy. Typically redundancy for a recovery action is either in time or in space. A ***redundant*** component is one that can be connected to the same inputs and can provide the same outputs as another component. If this component is always connected to the inputs, the failover process is simply to throw the output switch from the currently active device to the redundant backup device (**switchover**). The state data is maintained by the input stream, so no interaction between the primary and backup would need to take place. This is redundancy in space (***spatial redundancy)*** because there is essentially no time needed to perform the switchover.

Another method of spatial redundancy is if the active component periodically captures the state and forwards it to a standby that is just validating the state and storing it. In the event that the active is no longer providing the service, a recovery action would be to restart the operation from the last known good state on the standby system. Both of these approaches can be referred to as active/standby.

To continue with the networking theme and carry forward from the isolation phase, re-routing and the acknowledgement or negative acknowledgement of TCP packets would be a recovery action as well. In this case the protocol is built to be a reliable transport so the negative acknowledgements, or lack of any acknowledgement, will cause the protocol to start the recovery action. This is redundancy in time, or ***temporal redundancy***.

The following sections address the reloading or restarting of an application, protocol or even the operating system. This typically requires graduated levels of time to do the recovery.

6.4.5 Techniques

Switchover. This technique is used in a redundant operation and can refer to the switchover of any component to another whether it is performing the same operation and only switches the output or that it recovers from a known state. The component in this case could be hardware or software. The redundancy can be of a peripheral component or even the core processing element. This typically is identified as a 2N redundancy.

Re-routing. This technique could also be referred to as load balancing or load sharing. This is used with an N + 1 or more generally N + M redundancy. This means that there is N components providing service and any one of those components could be replaced by M other components.

Software Rejuvenation. The simplest form of this technique is by the example of a screen refresh — it rejuvenates the image. In the general case this would be the restart or reloading of an application, dynamic library or protocol to cause a system to reset and initialize the resources it uses to perform the service.

Reboot. This is the most radical of the rejuvenation steps. This restarts the operating system as well as the protocols, libraries and applications.

6.4.6 Dependencies

Recovery depends on the policies, redundancy elements and the techniques which are typically part of set of policy descriptions that are used when the system is configured. It is the last of the process steps for service restoration.

6.5 Repair

6.5.1 Introduction

Repair, in a live system, requires some form of hot replacement. Again, a system must be designed to support this activity. To repair a failed component, a replacement is hot inserted, powered on, connected to the bus, validated through off-line diagnostics, and configured.

6.5.2 Objective

The objective of this process is to return the system to its original capabilities including levels of redundancy.

6.5.3 Concepts

Physical Replacement. Repair is the replacement of the defective component. This phase is generally designated for the operator (human) assisted portion of the process. This is usually the most time consuming portion of the process.

Download. Once the failed component has been replaced and validated as operational the process of configuration is necessary. This typically consists of loading software into the replaced module and activation of a driver.

Dynamic Reconfiguration. This is a way to reconfigure by adding new components or removing old without impacting services running on the system that are not using the components. This is addressed in Section 5.0.

Off-line Diagnostics. In order to repair a component it may be necessary to run additional diagnostics which could be run while the component was active. Additionally, these diagnostics are run before putting a new or repaired component back into service to help ensure that component is functioning properly

6.5.4 Approach

Performing a repair operation depends on the nature of the failure that was corrected. The normal repair action would be to replace a component in the system. If this is physical hardware, the repair craftsperson should identify the defective component and replace it. Diagnostic tests should be run to test the new component, without affecting the running system. Then, the system should be configured. The process could include activating the system as the spare, or redundant component, activating load sharing, updating routing information, or even activating the new component as the active by forcing a switchover. These are all case-by-case actions that an individual implementation may choose and are supported by the management framework.

In the case of a software component, the failure could be as easy to repair as downloading the software again. More than likely the problem was not that a local copy of the component was damaged or destroyed but the user was not aware of a specific sequence of events that caused it to fail. This can be corrected through a system patch. Patches are typically small changes in the code, preferably while the system is performing or using the component. A more encompassing action would be to replace or upgrade the software. The upgrade process is described more specifically in Section 5.6. A successful software upgrade requires the ability to precisely identify the version that is running in the system. An upgrade requires understanding the software's interactions with other components in the system (dependencies) and the conversions of any data storage that may need to be updated as well. Also, a software upgrade should include a rollback feature that allows the system to be returned to the original operation prior to the upgrade.

Diagnostics are tools for verification. The final step in the repair action is to be sure that the new component is working properly.

6.5.5 Techniques

Component Replacement. This technique includes physical replacement of the failed component.

Debug and Diagnostics. Before a component is removed from a system diagnostic and debug utilities may be used to find the atomic component that has faulted within the component. These techniques are also used just after a component is placed back into a system to verify that the component works correctly.

Off-line diagnostics are typically provided by the manufacturer of the component. A method needs to be in place to connect an off-line component to an alternate input and output stream in order to run the diagnostics.

When off-line diagnostics are complete, the information about the diagnostics (including time, version, results, etc.) should be stored in the system information data block for that component and/or its subcomponents.

Software Patching. This technique involves replacing pieces of a software component. The desire would be to do this while the component is being used. A ***residual signature*** is needed to show that a patch has been applied. A mechanism to remove the patch and its associated signature must also be provided.

Software Upgrade. This technique involves the replacement of the component and dependent components in the system. The operation should be performed while the system is providing service but will normally take some level of switchover action and conversion action after the installation is complete.

6.5.6 Dependencies

The repair action depends on the manual actions of the craftsperson. This is the most error-prone item and the system must be prepared to expect the unexpected. Other dependencies include component identification and versioning, and diagnostics.

6.6 Notification

6.6.1 Introduction

Components within a system must interact with each other to enable fault management. Notification of fault information and the progression of the fault management process may occur through various communication interfaces. For purposes of fault management, the notification function focuses only on communication capabilities and interfaces between the fault management processes and the fault communication between the layers.

This function builds upon the concept of system component interfaces (Section 5.4), and management interfaces (both internal and external) that are detailed in Section 5.5.

6.6.2 Objective

Notification is a key capability of the fault management process. The objective of notification is to enable management middleware and other system components to access fault reporting, state change performance and status information that could proactively predict faults.

6.6.3 Concepts

Notification may include information context and content on:

Autonomous Notification. Notification information is automatically generated and communicated through the component's interface.

Directed Notification. Based upon the system design notification information and fault management control information can be directed to specific interface(s). These interfaces may be inter-layer; intra-layer, or external.

Indirect Notification. An indirect notification of a fault may be from a higher entity, or from a peer component using the techniques described in Section 6.1.5.

In-line Notification. Non-faulted components immediately adjacent (up and down) to a detected fault should typically get an in-line notification of the fault. For instance, a break in network cable or connector will typically be sensed by the ***Media Access Controller (MAC)*** in the hardware layer. This MAC could immediately communicate this information to the driver via an interrupt, or could respond with an interface unavailable error return code in response to the next I/O request. Both of these responses would be in-line notification. Function return codes are another example of 'in-line' notification. Examining the return code provides immediate information to the calling process on the failure, status, or success of the operation.

In-band Notification. Notification of events, faults, or exceptions may be reported using the same framework, protocols and hardware as other inter-process messages. These messages are considered ***in-band*** as they use the same "band" as other inter-process messages. Notification messages may need to be sent several times during a recovery process. For instance, in the recovery from a fault, redundant components transition from a standby to an active role assignment, load a copy of a driver or checkpoint data and may encounter exception conditions. Each of these occurrences may be cause for a notification.

Out-of-band Notification. Notification of events, faults, or exceptions may also be reported out-of-band as directed communication with one or more of the management interfaces. For instance, in the recovery from a fault, redundant components may transition from a standby to an active role assignment, load a copy of a driver or checkpoint data, or encounter exception conditions. While still reporting state and error conditions in-band, the same information could be reported as out-of-band to the system log/console, middleware and other management interfaces.

Management Interfaces. As described in Section 5.5, these interfaces can be interprocess, intra-layer, layer-to-layer, local, or remote.

Administrative Notification. When available, fault management notification should be reported on the system console. Administrative notification might also consist of activating LED's, alarms and relays. The administrative interface can also provide notification information in the fault management process; acknowledging or silencing an alarm or using the extractor handles to light the blue light in CompactPCI systems, provides event information that is visible (or audible) without having to use the standard system management console.

System Log. Event information, exception conditions, state changes and context information should be reported to and recorded in a structured event log, such as a system log.

6.6.4 Approach

State changes (whether generated by faults or not) of hardware and software *resources* within the defined system model may signify increased or diminished capabilities, and should generate immediate autonomous notification messages.

Detection of a fault should generate an immediate autonomous notification message. Depending upon the type and severity, the message could range from an entry in the system log, a report on the system console, or as a alarm to the middleware layer or to one or more of the management interfaces (Section 5.5). As described in this section, the notification message may be delivered as a best-efforts one-way communication, or as a two-way acknowledged, persistent, or periodic communication. Likewise, any unresolved fault that straddles more than one layer of the system should be reported both in-line to related component and out-of-band to appropriate management interfaces, based upon its type and severity.

After a fault has been detected and reported, the middleware and other management functions may use component interfaces in a directed manner to invoke the fault management process. This is typically used to sequence the fault management process, and to isolate propagation of the fault.

In general, system components with a notification capability should retain this fault, status, performance and state information until: a) cleared by a directed command or by an acknowledgement; b) capacity is exceeded (may be none, indicating no capacity, or may be some form of wrap-around register/buffer); c) notification that system fault has been repaired. This concept could apply to a communications controller chip, a driver with state and error registers, a protocol stack, kernel, system log, or application.

In the fault management process, it is typical to use request/acknowledge, or other forms of reliable communications methods, as available. After the initial report of a fault, it is not immediately known whether the notification reports the original fault, or a subsequent fault that has resulted from the original (as of yet) undetected fault, and whether any other components may have faulted because of the original fault. Using directed, reliable communications methods in the fault management process ensures that undetected faults are recognized. A failure to respond to a directed-acknowledge request within a reasonable period of time becomes a method of detecting failure. It is important to understand that a failure to respond could indicate that: the responding unit is faulty, the communication link is unavailable, or that the reception path on the original sender’s interface is faulty.

6.6.5 Techniques

“Go out with a bang, not a whimper”

Whenever possible, abnormal termination of software components or the OS should be recorded in as much context and detail as possible to allow for diagnosis and debug of the fault. Typically this type of information is found in application core and OS crash files. Source file information is usually needed to fully debug this information.

Information Context / Content

The content and context of the notification should be appropriate to the management interface. A non-recoverable media fault reported from the I/O driver to the calling thread would typically be limited to *return code* error information (lightweight). While on the system console, the notification of the same event might indicate that it was a disk error during a read (this state information was unnecessary in the context of the affected process since the calling thread knew whether it was a read or write operation) was encountered at time and date. Even more information might be recorded in the system log (heavyweight), and terse event notification might be sent to the middleware and external interfaces, assuming that if they wanted more information and context that they would make a directed or polled request to system log.

Levels of Fault Management

Implicit in this system and fault management model is the concept of multiple levels of fault management that may be implemented in each of the layers and have different objectives, requirements and responses to the same fault conditions.

In general, if a layer capability can autonomously handle the fault, it should independently do so while reporting the fault and recording the fault management actions. If it cannot autonomously handle the fault, one of the following conditions may be the cause:

- the fault's type or severity is beyond the layer's capability
- the autonomous fault management capabilities of that component have been explicitly disabled by a fault management entity at a higher level (explicit prohibition)
- the pre-set error thresholds have been exceeded (conditional prohibition)
- the autonomous fault management capability is either not present or not functional because of temporal or resource conflicts with other layers and service-oriented priorities

There is also more capacity, system knowledge and intelligence at higher level management entities. However, the latency of the recovery from the fault also seems to grow as more layers are involved.

Using the disk error example started above, let's examine the layers of fault management and notification in a disk read error. In general, errors that occur while reading file system data structures are propagated all the way up to the original request. Any request to write to a file system data structure that fails because of a write or read error could result in an inconsistent file system. A file system that can no longer be accessed because of errors can be unmounted and subsequently remounted after the reason for the errors has been removed (perhaps a power failure or bad connection) and the file system is checked.

All device errors are reported from the lowest possible level (the device driver or device manager) to the console, system log and/or to the middleware management layer. It does not make sense to report the error at every step of its propagation to the user program. The original cause of the error is the most useful information to a system administrator or management interface.

More specifically, after the initial fault at the hardware layer, the controller/drive might retry x-times to eliminate transient and simple seek errors; then it might adapt the skew to attempt to recover the requested block; devices with ECC, parity, advanced data encoding, mirrored drives, and RAID-like devices with striping might recover the data automatically, repair the fault by relocating the information without reporting an in-band error. However, the detailed context of the error and recovery might be communicated and captured as warning information. Based upon a frequency or rate of change threshold, this type of warning might become a stronger alert and then an alarm notification to the layers and management interfaces above it.

If the disk read condition is passed up to the I/O driver in the OS layer, the driver might attempt its own form of error recovery, perhaps resetting the controller and trying again. ***Hardened drivers*** check error codes and will eventually time-out and report an error. If the driver does not resolve the error in a determinate period of time, it would communicate an error that would be reported and captured.

At the OS layer, if the device is a virtual device, the OS might attempt its own recovery by replicating the request upon another interface. If still unresolved, the disk read error would be passed to the calling thread as an error return code, captured in the system log, and reported as a fault to at least the middleware management interface.

At the application layer, the return code will probably invoke some form of error trapping and processing to prevent the propagation of this fault to other software components.

At the middleware layer, the notification of the fault event will start an appropriate fault management process, (try reading or writing to another file, unmounting and checking the disk) which might include the escalation of this event to a higher level in the form of a trouble ticket or repair request.

6.6.6 Dependencies

Notification is dependent on the components that are handling faults to generate the appropriate messages. It is also dependent on the communications and messaging systems to get a message from the component which is handling a fault to the components that need this information.

6.7 Prediction

6.7.1 Introduction

Prediction is the process of observing the operation of the system and determining when a component will need to be replaced, repaired or subjected to further diagnostics.

6.7.2 Objective

The objective of prediction is to reduce the occurrence of faults by preemptive notification and repair.

6.7.3 Concepts

- **Data Collection.** Health data must be collected in the system model to make it possible to predict faults.
- **Data Analysis.** Models of the types of failures

6.7.4 Approach

Fault prediction uses periodic or historic information gathered about a system and its components in an attempt to determine when and where a fault is most likely to occur. The data accumulated about the specified components or subsystems might entail previous failure information, device monitoring data, MTBF statistics, and applicable data gathered from associated components. Using the data collected from the sources at hand, a statistical probability of failure can be derived.

Some examples of data that could be collected are the packet and error rates of network connections, the temperature or speed of components, or watermarks for OS parameters. Using network rates, one can derive if a net segment is overloaded or beginning to fail. Temperature and/or speed of hardware components can indicate either immediate failure, as when CPUs and other ICs overheat, or the need for preventative maintenance, as when fans slow down or disk drives either slow down or heat up. Watermarks on memory, stack, or process use in the OS can indicate that a process is stuck or out of control. All of these items can give indication of a pending failure, with significant time to avoid the failure and maintain service availability.

6.7.5 Techniques

Trends

Prediction can be done based on trends of a single variable, but frequently requires looking at two or more variables. For example, CPU temperature trends would need to be viewed in light of airflow and inlet air temperature to be useful for trending.

Multivariate Correlations

As noted above, most trends for failure must be viewed as composites of several variable. For example, network re-try rate is meaningless without a network load factor.

Expert Systems

After a number of faults have occurred in a system it is possible to analyze the faults for trends and use expert systems to watch for these trends to occur again.

6.7.6 Dependencies

Prediction is dependent on the system model collecting the required information and on the system designer setting up sufficient detectors.

7.0 Open-Architecture Systems

The preceding sections have described the requirements of and an architectural approach to building high availability systems. However, the goal of the HA Forum is not simply to describe how to build high availability systems, but to describe how to build ***open-architecture*** high availability systems.

This section will explain why high availability systems have not traditionally been open systems, discuss the relevance of the open-architecture, and identify the basic open-architecture building blocks for open-architecture high availability systems. Subsequent sections will then discuss each of those building blocks in more detail.

7.1 Open Architecture and High Availability

Computer systems are complex; they contain many different elements of technology (e.g., hardware, operating system software, application software) which must work together in precise ways to produce any useful results. As a natural consequence of this complexity, technology in early computer systems was carefully designed *together* to accomplish some specific purpose. The entire early computer industry consisted of vertically integrated computer manufacturers that produced these carefully integrated systems.

With the advent of open-architecture systems, this situation has changed radically. The norm today is for multiple technology providers to contribute subsets of the entire technology stack which makes up a computer system. One company makes CPU chips, while another builds computer hardware platforms which incorporate those chips, a third produces operating systems, and so on, with multiple competitors vying for market share at each of those technology tiers.

In this environment, standards are critical. Each implementation of a technology building block must provide common sets of services, and must deliver those services in a predictable way, so that other building blocks, created by other companies, can interoperate with any of them. For example, it is important in a PC environment that hardware manufacturers all make hardware platforms which deliver similar services and interfaces so that an operating system such as Microsoft Windows or Linux can run on any of them.

Building high availability systems is more complex than building traditional computer systems. Not only must high availability systems deliver all of the same services as traditional systems, they must also manage themselves to provide some level of dynamic reconfiguration in the face of faults. This added complexity begins at the hardware platforms, and has resulted in most high availability systems to date being built as vertically integrated systems, rather than as open-architecture systems.

The common industry standard computer hardware platform, a PC, does not include key hardware capabilities needed in many high availability systems such as redundant internal data paths, platform management, hot-swappable components, etc. As a result, manufacturers of high availability systems have historically built non-standard hardware platforms, which then required non-standard operating systems, and often even non-standard application software.

Today, however, standards are emerging that can enable the same sort of open-architecture technology building blocks, which has revolutionized the traditional computer industry to be applied to high availability computer system design. Thus, the HA Forum has been chartered to foster this industry development by identifying and creating relevant standards which will allow the building of high availability systems from open-architecture building blocks.

7.2 Open-Architecture Building Blocks for High Availability Systems

The first key step in creating the ability to build high availability systems from open-architecture building blocks is to identify what those building blocks are. Each building block will provide one part of the overall technology stack in a high availability computer system. Ideally, each building block will be independent of the others so that multiple, competing products can be marketed for each building block, with interoperability between the building blocks enabled by the adoption of common interfaces for common services.

Since there is a well developed model for open-architecture building blocks in traditional (i.e., not high availability) systems, it is natural to begin there. This model has clearly defined building blocks for hardware platforms, operating systems, middleware packages (e.g., communication protocol stacks, database systems), and application software. High availability systems are built out of this same set of technology, with the added requirement that managed redundancy somehow be a part of the overall system.

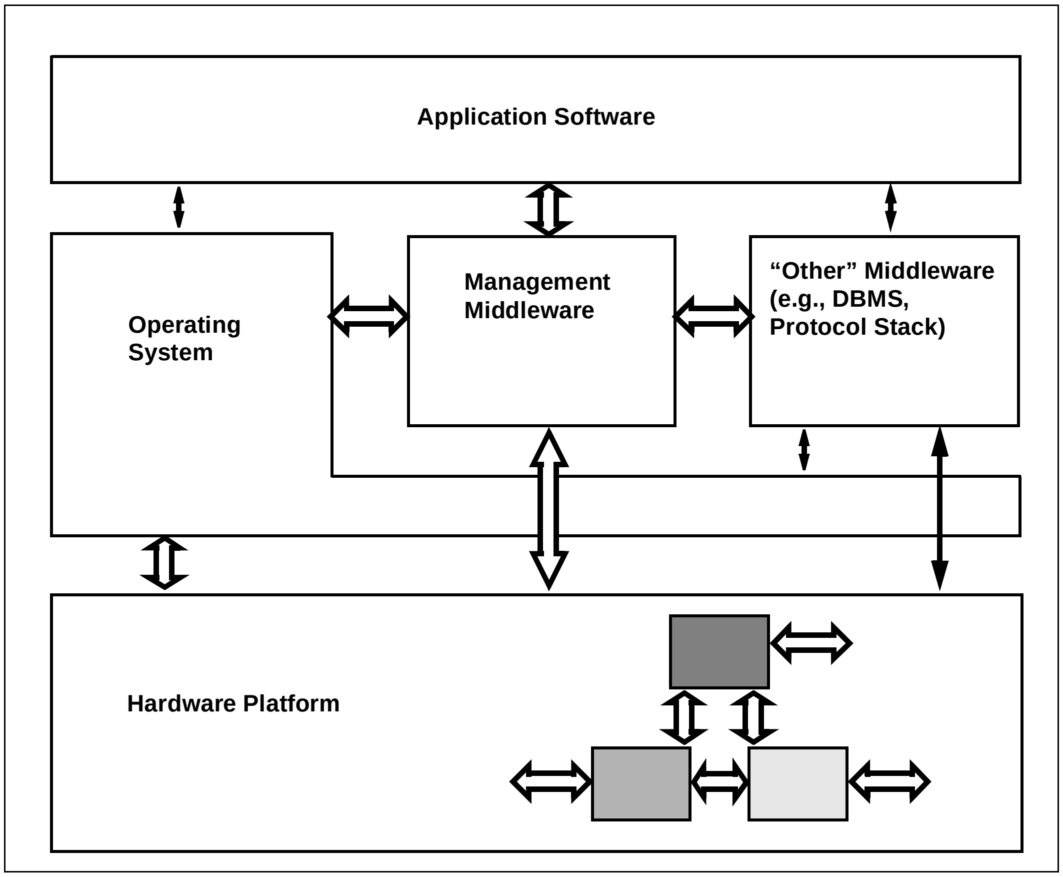
A major technology differentiation of high availability systems is the inclusion of configuration and fault management functions. As described earlier, this requires the detection, diagnosis, isolation, recovery, and repair of faults anywhere they could result in the failure of the high availability system. The approach taken by the HA Forum is to define this function as an incremental addition to the existing open-architecture model for computer systems.

Thus, a single, new building block has been defined, called management middleware. This is defined as a set of configuration and fault management capabilities which are independent of any particular hardware platform, operating system, or other technology building blocks. While it is possible for operating system vendors, hardware platform vendors, or application software vendors to provide this new capability, defining it as a standard, portable function with standard interfaces to the other building blocks which make up a high availability system maximizes the ability to mix and match technology building blocks throughout the system.

Figure 13 depicts the technology building blocks that should be available from multiple, competing sources if high availability systems can be built from open-architecture technology. In this figure, each shaded block represents a piece of technology (building block) which is relevant to building high availability systems, and identifies (by the wide gray arrows) interfaces which are relevant to high availability capabilities. Ideally, each building block could have multiple, competing sources, and be exchanged without impacting other building blocks. In reality, this ideal is not likely to be met perfectly; but, the more similar the interfaces are between building blocks, the more open the system becomes.

The remainder of this section will describe each of the building blocks and interfaces depicted in Figure 13.

Figure 13. Open Architecture Building Block in an HA System



The ***hardware platform*** consists of the entire set of hardware, firmware, etc., normally provided by a hardware system vendor, ready to support an operating system like Windows or Linux. For the purposes of building a high availability system, of particular significance is the platform management infrastructure which permits fault-management operations on the components in the hardware.

The small boxes within the hardware platform box indicate that a further degree of open architecture is desired within the hardware of a system. In addition to providing a platform that supports the right interfaces to operating system and other building blocks, open systems also provide a capability to integrate multiple, third-party peripheral cards, power supplies, and potentially other components. Thus, the interfaces between these components, as well as the interface between them and other vendor-specific features in the hardware platform, need standardization. Again, in particular for high availability systems, management interfaces are important considerations for standardization within the hardware platform.

The ***operating system*** provides basic process scheduling and resource control for application software and middleware. Part of the resource control function is the provision of device drivers for software access to hardware resources. In this model, the operating system provides the same sorts of services required in systems that do not have high availability features. That is, additional services that are specifically designed to provide fault-management of the system as a whole are considered part of the management middleware building block in this model – even if they are provided by an operating system vendor as a part of their product. This is why the interface between the operating system and the application software and between the operating system and the other middleware are shown as narrow arrows (indicating that it is not an interface that has significance to the high availability capabilities in the system). However, even with this restrictive view, the operating system will require certain capabilities to operate in a high availability system. These include exporting to management middleware an interface for management of the operating system itself, and the ability to deal with a changing (hot-swappable) hardware configuration as required to support the configuration management capabilities described in Section 5.0.

The ***management middleware*** contains the new system-level functionality that is unique to high availability systems to provide fault management and configuration management capabilities. While it is not required that this be separately developed and integrated into the system, by defining this as a separate building block and standardizing the interfaces to the management middleware from all other building blocks, it becomes possible to achieve the open-architecture high availability systems much more quickly. Note that the interface to the hardware is shown going through the operating system. Access to the hardware is handled by device drivers in the OS but the semantics of the interface are defined by the underlying hardware.

The ***other middleware*** consists of software packages which add additional capabilities to an operating system such as database management or communication protocol processing. Middleware often contains direct interfaces to the operating system, to hardware devices (accessed through the operating system via device drivers – indicated by the arrow passing through the operating system level to the hardware platform), and to application software. To be incorporated well into a high availability system, middleware packages should also export a management interface for fault and configuration management within the middleware.

The ***application software*** may or may not be aware of the high availability system infrastructure. To support the cases where it is, there must be an interface between the application software and the management middleware. This interface makes the application itself manageable, and may provide access to services unique to the high availability system such as checkpointing application state or heartbeating. Other than this interface, application software will generally interact with the rest of the building blocks (operating system and other middleware) no differently than in non-high availability systems.

8.0 Layer-Specific Capabilities – Hardware

High availability hardware system architectures are created by combining fault domains into service groups in such a way that the system can continue to operate even when any particular fault domain is out of service. A wide variety of fault domain configurations are possible. Roughly speaking, high availability system architectures fall on a spectrum based on the granularity and complexity of the fault domain model. At the two ends of this spectrum are:

Clustering. A fault domain consists of an entire computer, complete with CPU, memory, I/O controllers, I/O devices, power conversion and distribution systems, cooling systems, etc. Multiples of these computers (often called nodes) are then used as redundant fault domains.

Hardware Fault Tolerance. A single computer is made up of multiple, redundant fault domains. The hardware design is such that the computer continues to provide full service even if any of its constituent fault domains fail.

Today, many high availability computer systems fall between these end points. These systems contain some fault domains which look and operate much like nodes in a clustering system, but have other fault domains that are managed in a fault-tolerant mode. An example of such a system would be two complete processing units, complete with CPU, I/O controllers, and backplanes housed in a single enclosure with common power supplies and fans, and connected to a single, shared RAID disk subsystem.

The required capabilities of hardware components are dependent on the partitioning of the overall system into fault domains. At the highest level, hardware capabilities of fault-managed systems can be categorized into three sets:

- Capabilities to allow continued processing after failure of a fault domain using redundant fault domains
- Capabilities to allow highly reliable (often redundant) communication among fault domains
- Capabilities to allow management of the fault domains that include fault detection, diagnosis, isolation, recovery, and repair at least to the fault domain level of granularity

8.1 Redundancy

The most fundamental capability provided by the hardware of high availability fault-managed systems is the provision of redundant ***fault domains***. In any high availability system, fault domains must be identified. Then, for each fault domain, the required redundancy must also be identified that will permit continued provision of the services of that fault domain when it is not functional.

As an example, in clustered systems, fault domains are complete computer nodes. Redundancy is provided by including at least enough nodes to support the minimum required performance of the system when any one of the nodes are out of service.

For systems that contain fault domains smaller than a complete computer node, each fault domain must be made redundant – at least in a N+1 mode, so that whatever services a particular fault domain provides to the system, those services will still be provided when that fault domain is out of service. Typical subsystems that make up fault domains, and thus are provisioned redundantly and organized into service groups include:

- Processing subsystems
- I/O controllers

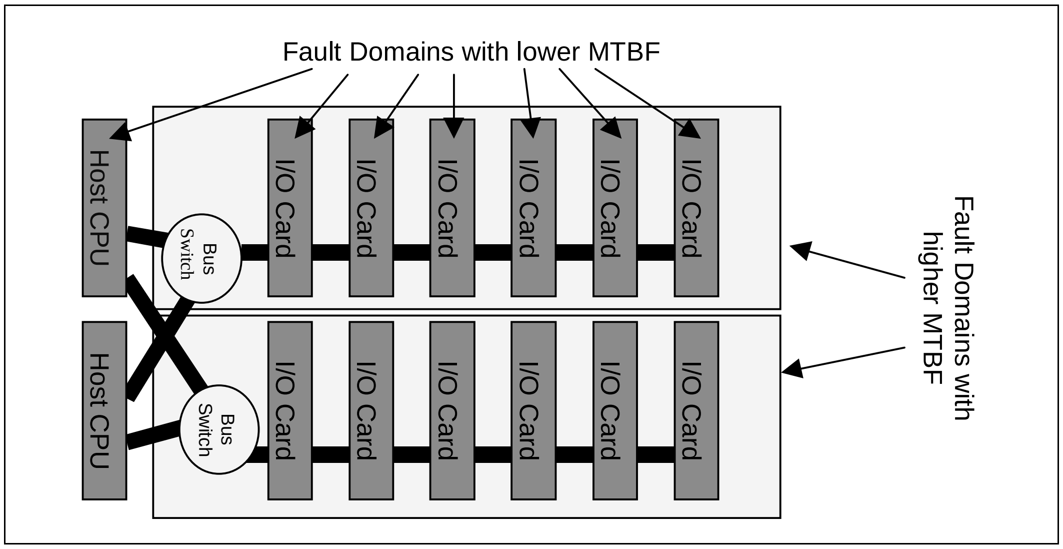
- Mass storage subsystems
- Peripheral devices
- Power supplies
- Cooling modules

8.2 Communication

The fault domains within a high availability system interact with each other to create a complete system. This interaction occurs through various communication mechanisms. For the purpose of a fault domain analysis, the communication mechanisms of significance are the ones between fault domains. Failures of communication paths *within* fault domains can be considered another fault, that can lead to the failure of the fault domain.

Note that systems often contain nested fault domains. For example, if there is a non-redundant communication path between a set of fault domains (and that communication path is critical to those fault domains providing required services), then that set of fault domains plus the communication path becomes a larger fault domain. Figure 14 shows an example of this arrangement. Such a system architecture may make sense if the MTBF of the communication path (an I/O bus in Figure 14) is much larger than the MTBF of the nested fault domains (the I/O controllers in Figure 14), or if the MTTR is much smaller.

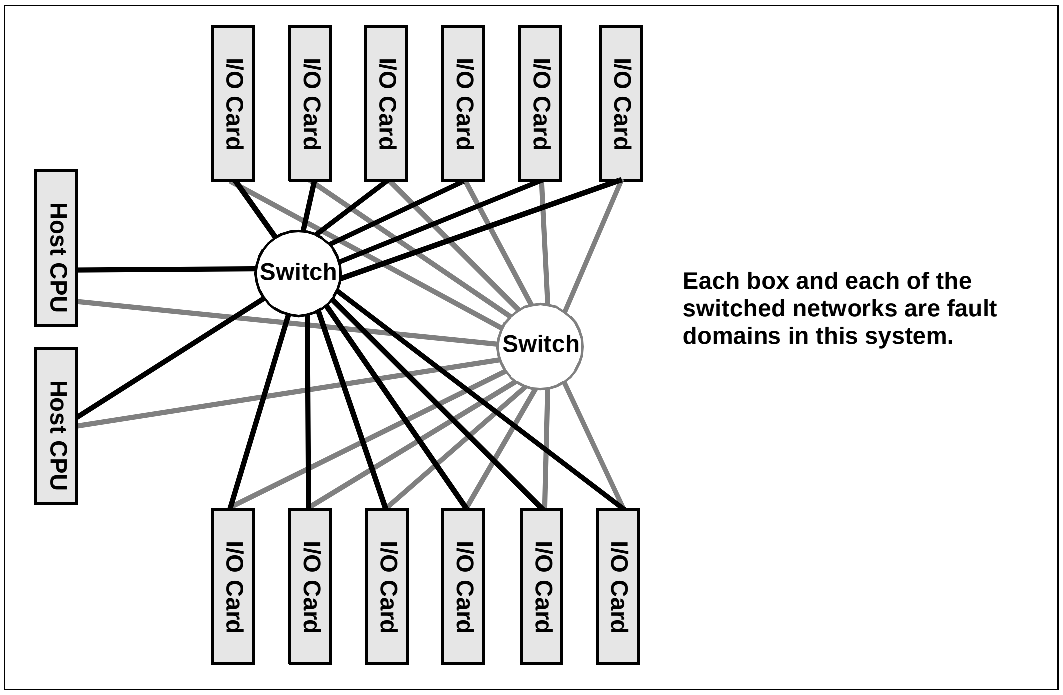
Figure 14. Nested Fault Domains



In contrast, Figure 15 shows a similar system where the bussed interconnects are replaced by a redundant switched network interconnect. In this system, each of the switched networks is a separate fault domain, but because each I/O controller and host processor are connected to both networks, the fault domains remain separate from each other rather than nesting.

For clustered systems, communications between nodes are typically via standard local-area-network connections, though these may consist of dedicated LANs used just within the cluster for performance reasons. Redundancy in the network connections between nodes is generally required so that the cluster can continue to function correctly if any one communication link or path fails.

Figure 15. Redundant Switched Interconnects


I/O Card
Host CPU
Switch
Switch
Each box and each of the switched networks are fault domains in this system.

For fault tolerant systems, communications among fault domains will vary depending on the specific characteristics of each fault domain. For example, current sharing power supplies communicate via monitoring the voltages and currents being produced or via a separate current share signal. Even more extreme, fans may communicate by jointly pressurizing a common air plenum that is engineered in such a way that airflow continues across all system components even if one of the fans fails.

Other fault domains, such as processing subsystems and I/O controllers, communicate through various data paths within the system. These may be I/O busses or point-to-point data paths. Because they typically are non-redundant, bussed interconnects define fault domains consisting of everything connected to a single bus, but bus failures themselves are generally very low probability. As a result, a common compromise in the design of hardware for high availability systems is to define a hierarchy of fault domains, as shown in Figure 14.

Because of the compromises required for this sort of configuration, next-generation system interconnects such as InfiniBand, RapidIO, or other switched fabric technologies provide higher levels of availability with the same amount of hardware by eliminating these larger fault domains, as shown in Figure 15.

Finally, communication paths to and from the high availability system itself may need to be redundant. In a sense, this is simply a redefinition of the system to the next higher level – i.e., to incorporate the devices connected to the system in question into the system itself. Of more relevance, the termination of a single data path will be either:

- to a single point, which will necessarily be contained within a single fault domain
- to multiple points within multiple fault domains

Because some communication links are difficult to terminate at multiple points in a system, and because redundancy in the external communication paths is desirable for its own sake, high availability systems are often designed with redundant external communication links, each of which is logically part of the fault domain that includes its termination point in the system.

8.3 Platform Management

As described above, high availability systems include redundant hardware, and this redundancy can be described as a set of service groups, with each service group consisting of redundant fault domains. The hardware of high availability systems can be implemented in a wide variety of ways, depending on system requirements. But, no matter how fault domains and service groups are designed, a common requirement of any system that includes redundant hardware is some level of manageability of the hardware platform. This capability is called ***platform management***. The function of platform management is to provide for monitoring and control of the hardware components. The minimum goal of platform management in a high availability system is to provide monitoring and control required for the detection, diagnosis, isolation, recovery, and repair of fault domains. Beyond that, additional platform management capabilities aimed at predicting and preventing fault domain failures may be included.

Platform management activity generally requires communication of management data between management software and hardware components, or among hardware components themselves. This communication can be provided in-band (i.e., using the same data paths used for the primary operation of the system), or out-of-band (i.e., using dedicated platform management data paths). Often, out-of-band communication is used in high availability systems, because the management data path can exist in a separate fault domain from the primary system data paths. This is important since the management function is often critically required after a fault has occurred to cause a system recovery. In the face of a fault, primary data paths may well be unusable, while separate management data paths can still function.

The minimum required platform management functionality may be delivered in a variety of ways; however, a subsidiary goal of defining standard interfaces leads to additional desirable capabilities of the platform management subsystem.

8.3.1 Fault Domain Failure Detection

However fault domains are constructed, a critical hardware capability is the detection of failures of fault domains and communication of those failures. This may be communicated out-of-band through a management data channel or in-band via unambiguous observable behavior (or non-behavior). A primary example of the latter is ***fail-safe*** behavior, where a fault domain contains a self-checking capability which causes it to promptly shut down when a fault is detected. The resulting shutdown is then observed by other parts of the system.

Beyond the immediate communication required for fault diagnosis and isolation, hardware fault domain failures must also be communicated to appropriate subsystems (or people) in order to trigger recovery and repair actions. For example, consider a failed current-sharing power supply — the immediate fault detection and isolation is carried out by current sharing circuitry and output diodes. This action is effectively transparent to the rest of the system. Thus, another means of communicating the failure must exist to trigger an action to repair the failed supply. This may be via de-asserting a ‘power supply OK’ signal or management event generation.

When a system contains fault domains that are effectively in a standby mode, there is a need for detection of latent faults in these domains. That is, if the primary failure detection mechanism is observation of normal operating behavior, the hardware may need to provide a separate mechanism for detection of faults in fault domains which are not normally operating.

8.3.2 Fault Domain Diagnosis

In cases where a failure of a fault domain is detectable, but it is not immediately evident which of several redundant fault domains has failed, hardware must support a diagnostic function which will determine where the actual failure exists. For example, consider a system with multiple fans, each of which makes up a separate fault domain, because the failure of any one fan can be compensated by increasing the speed of others. If a failure of a fan is detected indirectly (e.g., temperature or air-flow detector) additional diagnosis may be required to determine *which* fan had failed so that the proper isolation, recovery, and repair actions may be initiated.

Diagnosis of failures may occur through diagnosis capabilities of the hardware components themselves, or through capabilities of management software that can analyze a variety of indications and/or initiate various actions in an attempt to determine which hardware component has failed. A diagnosis capability in the hardware itself basically means that it is able to answer the question, “Are you okay?” If the hardware cannot directly answer this question, then it must at least have the capabilities needed by management software for it to able to diagnose the failure.

In a system with nested fault domains (e.g., Figure 14), diagnosis may aid in converting a detected failure of the larger fault domain to a failure of the nested fault domain. For example, in Figure 14, if one of the I/O controllers on a shared bus failed in a way which caused the bus to hang, the fault domain associated with the bus segment would fail, which is what would be detected. It would be highly desirable in this case to be able to do a diagnosis of the individual I/O controllers, determine which one is failing, and isolate it from the bus. This would result in removing the fault from the larger fault domain, and restoring the functionality of all the other I/O controllers on the bus segment.

Additional diagnosis capabilities can also be useful to be able to repair a fault domain more quickly. For example, if an I/O controller fails, after the system isolates and recovers from the failure, it would be desirable to have the platform management system order the controller to run a self-test to determine if the failure was transient or permanent. If transient, then the I/O controller could be immediately reintegrated. If permanent, then a technician would have to be dispatched to replace the board.

8.3.3 Fault Domain Isolation

The ability to isolate fault domains from the system is critical to the design of a fault managed system. Isolation means taking whatever action is needed to prevent a fault from affecting other fault domains.

Often, this capability is an integral part of the design of a hardware component. For example, power supplies include circuits that monitor output voltages being produced, and quickly shut the supply off if voltages venture out of specifications. In other cases, another part of the system may need to initiate an action to isolate a fault domain. Even when isolation is automatic by design, it can be highly desirable, as a further safety feature, for any fault domain to be able to isolate itself from the rest of the system upon request from a platform management system.

Because of the importance of domain isolation, and because of the difficulty in guaranteeing the behavior of a fault domain which is already misbehaving, there may be multiple levels of fault isolation capability in systems. For example, if a host processor is not responding, it may be ordered to execute a system reset operation. If this still does not clear the problem, it may be ordered to power itself off. Similarly, if a particular I/O controller has failed in a system, it may be ordered to isolate itself from the I/O bus. If this does not work, a second level of isolation may be to isolate a slot on the backplane, or even an entire I/O bus segment (at the point of a PCI to PCI bridge, for example).

8.3.4 Fault Domain Failure Recovery

Fault Domain Failure Recovery means having the system recover from the failure of a fault domain; that is, to continue to provide its required service while no longer using the services of the failed fault domain.

The basic hardware capability needed is the ability to be reconfigured as required for the system to continue to provide its services using just the remaining functional fault domains. The specific requirements of the hardware will be dependent on details of the system redundancy strategies employed. It is possible that no special hardware capabilities will be needed to support recovery actions.

Examples of the sorts of hardware capabilities which *may* be needed are:

- Reprogramming a spare Ethernet controller MAC address so it can assume the role of a failed controller
- Reprogramming a switched fabric routing network to bypass failed nodes
- Switching a PCI to PCI bridge from non-transparent to transparent mode so that a slave processor card can become a system controller
- Selectively enabling or disabling hardware components
- Triggering a hardware condition which in turn triggers an automatic failover to a functioning fault domain
- Reassigning a boot device and forcing a reboot to load a specific configuration

As systems recover from failures of fault domains, a common problem is the preservation and/or migration of system state information in failed system components. In some cases, the duplication of system state information between hardware fault domains may be a function of the hardware platform itself (via, for example, a shadow memory capability).

More often, system state information is transferred to backup fault domains via management middleware and/or application software. Even in these cases hardware may support this operation by providing specific capabilities to perform hot, warm, or cold restarts. These designations suggest how state information in a system is preserved or destroyed as a result of a system reset operation. While there are no common formal definitions of these terms, a typical set of capabilities a particular system might provide could be:

Hot Restart. System reset operation clears and masks all interrupts, sets program counter and all CPU registers to pre-determined values, does not alter any system RAM, and begins operation.

Warm Restart. System reset operation clears and masks all interrupts, sets program counter and all CPU registers to pre-determined values (perhaps different from hot restart), does not alter certain parts of system RAM containing in-memory databases, and begins software reboot operation.

Cold Restart. System reset operation clears and masks all interrupts, sets program counter and all CPU registers to pre-determined values (perhaps different from hot or warm restart), clears all system RAM, and begins software boot operation.

8.3.5 Fault Domain Repair

One of the most complex features of high availability systems is the need to repair failed fault domains while the system continues to operate. To support this, the specific capabilities required in the hardware are dependent on the design of the fault domains. Because of the complexity, it is not unusual that the requirement for on-line repair of fault domains is a major driver in the system architecture, and the identification of fault domains in the first place.

In general, the more different sorts of fault domains a system has (i.e., the more the system design is a fault tolerant one vs. a clustering one), the more specific hardware capabilities will be required.

The capabilities needed to support on-line repair have several dimensions. These include:

- the basic ability to hot-swap fault domains
- notification to other parts of the system (e.g., the operating system and management middleware) that a system configuration has changed
- guidance to service personnel to help ensure correct repair actions
- on-line firmware upgradability
- maintenance of a system inventory

A few comments on each of these follow.

Hot-Swap

To support repair actions while the system remains active, the hardware must allow for the physical removal and replacement of a fault domain while the redundant fault domains which are providing service remain active. Furthermore, the hardware needs to be constructed to make this operation as failsafe as possible. Even if repairs are carried out by trained technicians, something as simple as dropping a screw can cause a system failure if the system is not designed with regards to on-line repairs from the beginning. Generally, high availability systems are designed to make fault-domain removal and replacement a very simple and safe operation.

Notification of Configuration Change

When a fault domain is repaired, the hardware platform configuration changes. At the least, a new resource is made available that the operating system and application program can begin using. This needs to be communicated to the operating system and/or application so that they can begin using the newly repaired fault domain.

The minimum requirement in a high availability system is to be able to cope with the removal and replacement of fault domains within a static configuration. However, it is highly desirable to be able to modify the system configuration more dynamically, adding additional hardware, upgrading hardware, etc. While this is not required for failure avoidance, it allows some amount of system upgrade to be accomplished without having to schedule system outages.

Guidance to Help Ensure Correct Repair Actions

One of the most common causes of computer system failures is erroneous operator actions. Performing on-line repair actions opens opportunities for errors. Therefore, a critical capability of fault managed systems is to guide repair actions to help prevent errors. This includes having common, standardized presentations of system alarms, designing the system to permit very simple repair procedures, and intuitive guidance through the repair itself.

Since the physical repair of a hardware domain will involve direct hands-on interaction between a system technician and the actual system hardware, having visual guidance for the repair action directly on the hardware itself is highly desirable. This often takes the form of LEDs and/or other small display devices that can be controlled through the platform management system.

On-Line Firmware Upgradability

Increasingly, hardware components contain field-programmable devices. This affords an opportunity to perform repairs and upgrades of the hardware without having to physically remove and replace components from the system. This can result in significant reductions in MTTRs, as well as eliminating opportunities for errors being made during a physical hardware swap. Thus, when programmable devices are used in a system design, having the capability to upgrade the firmware is highly desirable.

Maintenance of a System Inventory

In any system where hardware components and configuration can change over time, it is critical to be able to answer the question, “What is currently installed?” This includes a discovery capability that can detect all installed hardware. Individually replaceable units should be able to report, at a minimum, what they are (including revision level), what options they contain (if applicable), and unique tracking numbers. Typically, this information should be made available to the platform management system.

8.3.6 Event Notification

Throughout all the capabilities of platform management, a common requirement is to provide notification of significant events to other parts of the system, most notably the management middleware component of a high availability system. As significant events occur (fault detections, reconfigurations, etc.), the platform management system should provide information as to what has occurred. Desirable capabilities of the platform management event notification function include:

- A common format of event messages for all events
- An asynchronous generation of event messages rather than requiring software to poll devices to learn of events
- A ***publish/subscribe*** style interface for communicating events from platform management system to interested software
- Automatic storage of events in non-volatile memory so they are not lost if there is no current event listener
- System-wide synchronized time-stamp on events
- Common classification system for events to identify severity, urgency, etc., of event
- Inclusion of enough information with an event to permit immediate fault management action as well as later analysis for the purpose of root cause analysis on system faults

8.3.7 Additional Useful Hardware Capabilities

The above platform management capabilities have been aimed at the minimum required to support fault detection, diagnosis, isolation, recovery, and repair of fault domains. Beyond this minimum, additional platform management capabilities of the hardware in high availability systems can be provided in order to predict faults and prevent them from occurring in the first place.

Typically, these will involve monitoring analog values that can reflect on the health of the hardware even when a fault has not occurred. For example, a fan may be slowing down, but still functioning within specifications. This may be indicating a bearing wearing out, and with this warning, the fan can be replaced before a fault occurs. In another example, monitoring a temperature may indicate an impending problem before any component has actually failed, triggering a response to bring the temperature back into a safer range before a fault occurs.

8.4 Open Architecture Solution for Hardware Capabilities

As has been described above, there are a large number of hardware monitoring and control capabilities required in a fault managed high availability system, but the specific capabilities which may be required will vary significantly from one system architecture to another.

To support this variety of requirements in an open-standard way, the platform management system itself needs to have certain key capabilities. Among these are:

- An industry recognized interface to the operating system and management middleware
- The ability to be self-defining. The platform management system should be able to identify what capabilities it has so that operating systems and management software can adapt to the specifics of a particular platform.
- Use of an industry recognized management bus for intra-system communication. This allows for the interoperability of managed hardware components from various vendors with the overall platform management infrastructure.
- An industry recognized standard for implementing and controlling hot-swap
- An industry recognized hardware interface for key platform components such as power supplies, cooling units, system boards, and peripheral boards
- An industry recognized interface for platform alarming
- An industry recognized standard for redundant intra-system interconnects

8.5 Standards

8.5.1 IPMI

An example of an open standard, which does a good job of meeting the above hardware capability requirements is the ***Intelligent Platform Management Interface*** (***IPMI***) specification. This specification is available at http://developer.intel.com/design/servers/ipmi.

IPMI defines an open-standard abstraction interface and protocol targeted at component level platform management, which supports the sorts of capabilities described in the preceding paragraphs.

IPMI also defines a standard interconnect to support communication between platform components within the hardware layer and between the hardware and operating system layers. Communication between components is provided by the Intelligent Platform Management Bus (IPMB), which is typically bused throughout the chassis to connect all platform components including cooling units and power supplies. The IPMI specification also defines standard communication channels between the hardware platform and the operating system. These communication channels allow the operating system, management middleware, and applications to access platform management capabilities of the hardware. Other key capabilities defined by the IPMI specification include:

- Out-of-band communications
- Inventory management through FRU information
- On-line firmware/software upgrades
- Notification services for service personnel, including remote access to annunciation devices such as LEDs
- Asynchronous notification of platform events
- Logging of platform events for fault diagnosis

8.5.2 CompactPCI

The PCI Industrial Computing Manufacturers Group (PICMG) has developed, and is continuing to develop a family of specifications which target hardware technologies for high availability systems. Of particular interest for consideration in high availability systems are:

- CompactPCI Core Specification (PICMG 2.0)
- CompactPCI Hot-Swap (PICMG 2.1)
- CompactPCI System Management (PICMG 2.9)
- CompactPCI Power Interface (PICMG 2.11)
- CompactPCI Redundant System Slot (PICMG 2.13, draft in committee)
- CompactPCI Packet Switched Backplanes (PICMG 2.16, draft in committee)

There are also two specifications that may be of interest to designers of high availability systems, but are focused on software rather than hardware capabilities. These are:

- CompactPCI Hot-Swap Infrastructure Interface (PICMG 2.12)
- CompactPCI Multi-Computing Specification (PICMG 2.14, draft in committee)

8.5.3 Open Standards for Platform Alarming

In some segments of the telecommunications industry, due to the large concentrations of complex equipment in, for example, telephone central offices, standards for platform alarming are considered critical and already well developed. A comprehensive compilation of requirements and objectives for the telecommunications industry is defined in two documents, which contain Bellcore’s and ITU’s views to meet typical Local Exchange Carrier (LEC) operator service requirements and ensure system availability. Maintenance requirements include trouble detection, service recovery, notification, verification, isolation, repair, and human-machine interfaces for monitoring system status and initiating maintenance functions.

The Bellcore maintenance requirements are defined in Section 9 of Telcordia’s Operator Services System Generic Requirements (OSSGR), available as specification TR-NWT-001148. This is generally accepted as the North American Standard for this class of equipment.

Internationally marketed products tend to follow ITU publications. The ITU, International Communications Union, headquartered in Geneva, Switzerland is the international organization within which governments and the private sector coordinate global telecom networks and services. ITU-T Recommendation X.733 defines the alarm reporting function, which is largely similar to Bellcore’s. ITU-T Rec. Q.821 (03/93) – documents the stage 2 and stage 3 description for the Q3 software interface – alarm surveillance, notification and actions. The Q3 object model is also an IEEE standard.

The Telcordia and ITU standards for alarming go well beyond defining hardware capabilities, describing a complete approach to fault management. A complete treatment of this topic is beyond the scope of this paper, but it is a resource that any organization working on standards for fault management should consider.

The OSSGR Section 9 Specification defines an alarm as “a trouble or immediate condition that has an immediate or potential effect on the operation of the operator services system, and requires some action by a craftsperson to restore normal operation or prevent degradation of service.”

There are three alarm levels defined:

- **Critical.** Alarm shall be used to indicate a severe service-affecting condition, which requires immediate corrective action, regardless of time of day or day of week.
- **Major.** Alarm shall be used to indicate a failure in a major redundant service such that a further failure would create a critical condition. These troubles may require immediate craftsperson attention to restore or maintain system capability.
- **Minor.** Alarm shall be used to indicate troubles, which do not have a serious effect on service to customers, or troubles in services that are not essential to the primary operation.

The alarm levels are exclusive, with critical taking precedence over major, and major taking precedence over minor.

Alarm handling strategy is also defined in these standards with three defined levels:

Unit Level. Software recognizes faults in both the hardware and transmitted data, and correlates these faults to see if multiple ports are seeing the same information and takes corresponding action. It may switch traffic to some protection hardware and then throw an alarm to the next higher level, which in this case is the system level.

System Level. At the system level, the faults are again recognized and correlated. That is, it may map multiple faults to a single action. The actions could include bringing on-line protection hardware, changing a cross-connect, or simply raising an alarm at the system level. It could include sending a message to the network level that an alarm exists.

Network Level. The network level replicates what is done at the system level where the faults are again recognized and correlated. It may map multiple faults to a single action that could include bringing on-line protection hardware, changing a cross-connect, or simply raising an alarm at the network level.

Each level can work independently at decision management and enforcing actions but is also required to interact/inform with all the other levels.

8.5.4 Interconnects

There are upcoming standards in interconnect technology that will become important in HA systems. Some of the more prominent ones are listed in this section.

Gigabit Ethernet

Gigabit Ethernet is derived from the original 10 Mb/s and the later 100 Mb/s Ethernet. The link layer protocol and the packet structure is preserved. Ethernet transceivers can drive copper cable or FR4, and the technology can be used for backplane interconnect. Today most Ethernet networks are implemented as switched fabric. Likewise, this implementation would be used for backplane interconnect. Switched gigabit Ethernet uses point-to-point connections between nodes. Switches are used to route Ethernet packets between nodes. Ethernet borrows its electrical signaling, just like InfiniBand, from Fiber Channel. Gigabit Ethernet uses a differential pair for full duplex transmission. This differential pair is quad bundled into a set of four full duplex differential pairs (8 wires in total) with each set operating at a quarter of the base (1 GHz) frequency. The total bandwidth is 1 Gbit/s.

Ethernet implements the two lower layers of the OSI Reference Model (i.e., physical and data link). The upper layers (networking and transport) are implemented typically with a TCP/IP or UDP software stack. Ethernet is a message passing architecture. Thus, all work is performed by passing messages between communicating nodes. The protocols and software stacks have been and are being used pervasively. Therefore, the software is mature and very well understood.

Ethernet switches attach to their respective control node processor through PCI. Thus, a Ethernet fabric can be viewed as a switched PCI; the PCI bus is not eliminated from the solution, but rather a switch fabric is added between the PCI busses on opposite ends of the connection in the host. Ethernet does not provide any control over Quality of Service (QOS).

InfiniBand

InfiniBand is a new serial interconnect technology that is being developed for interconnecting systems (servers, storage devices, networking devices, etc.) in the server data center. Up to now, this level of interconnect within servers was performed by the PCI bus. The interconnection of the data center elements is referred to as a System Area Network (SAN). Historically, InfiniBand resulted from the union of the Next Generation I/O (NGIO) and Future I/O efforts. The InfiniBand specifications are developed by an industry consortium known as the InfiniBand Trade Association. This makes InfiniBand an open, publicly available standard.

InfiniBand uses a fabric topology based on point-to-point connections between nodes and switching elements. Interconnected devices, or nodes, send data that is routed through switching elements — much like a communications network. InfiniBand describes a three-layer architecture comprised of a physical layer, a data link layer, and a transport layer. Data is transported across the physical and link layer as packets. Messages flow across the transport layer and represent end-to-end transfer of data and control. Transactions (end-to-end operations) are initiated and controlled by the software via InfiniBand’s transport layer services (called verbs in the specifications). InfiniBand employs a message passing architecture to perform transactions. The fabric nature of the topology makes it easy to add nodes. The point-to-point connections ensure that the full bandwidth of the link is available as new nodes are added. This allows effective scalability.

The signaling is low-voltage differential with duplex dual-differential pairs per single width link. InfiniBand, as well as gigabit Ethernet, borrow their electrical signaling from Fiber Channel. Wider links (x4 and x12) are also supported. InfiniBand provides very high bandwidth (2.5 Gb/s) per connection. Host computers nodes attach to InfiniBand via host channel adapters (HCA). Other nodes attach into InfiniBand via target channel adapters (TCA).

Switches are used to route packets between nodes within a subnet. Subnets are topologies that can be address within the scope of a 16-bit local ID (LID). Routers are used to route packets between different subnets. InfiniBand offers a very rich set of transport, addressing and protection features. As much of the work within InfiniBand is performed in the software, InfiniBand is not PCI software transparent.

Rapid I/O

Rapid I/O (RIO) is a new interconnect technology that is being developed as a PCI replacement for board and backplane interconnect. The RIO Trade Association controls and develops the specifications. Thus, RIO is a open, publicly available standard to all trade association members. Like InfiniBand, RIO is fabric based. Thus, switches are used to route data between end nodes. Unlike InfiniBand message passing architecture, RIO uses a load/store base architecture analogous to PCI. This in conjunction with its physical addressing makes RIO PCI software transparent; RIO can work on existing operating systems with existing drivers as a direct PCI replacement without requiring modifications to these software elements.

RIO uses low voltage differential signaling (LVDS). (LVDS is an IEEE electrical signaling standard.) The clock frequency is 500 MHz with the data being double pumped. The data interface widths are 8 bits, 16 bits, or 32 bits.

PCI-to-Fabric Bridging

Technologies are available to bridge switching fabrics to PCI buses. Several of these are proprietary, but headed toward standardization. These systems typically have a fabric-based topology that use point-to-point connections between nodes. Two types of devices are needed for these types of systems: bridges and switches. The bridges provide a PCI bus on one side and a fabric connection on the other side. Switches provide multiple connections to nodes, which are usually bridges connected to the PCI bus. Packets are transported between the nodes via switches.

These type of systems can be made almost PCI software transparent, or they can be implemented as transparent with enhanced features. While PCI transparency is good for legacy devices, it can hinder future performance. Eventually, there will most likely be standards set in this area.

9.0 Layer-Specific Capabilities – Operating System

The ***operating system*** hosts applications and provides process scheduling and resource control for applications, middleware and device drivers. An HA-aware OS provides typical OS services as well as services that are specifically designed to provide fault-management capabilities either directly, or by escalating information to other layers for fault management and resolution.

At the OS layer, service availability can be enhanced by capabilities that improve system reliability. This section describes incremental software capabilities that can be used in the OS and applications to improve the effective-MTBF across a variety of HA system configurations. The term ***robustness*** is used to describe these additional fault avoidance and reliability capabilities at the OS layer. Similar design considerations may be used in the development of HA applications. The OS also isolates, prevents the propagation of, or masks the impact of potential hardware and software faults. These protection facilities help prevent errant applications and faulted hardware from bringing the entire system down.

Another area of HA-specific OS capabilities is the support of ***dynamic reconfiguration***. In a system model where redundant components are expected to fail, spare units must be rapidly enabled to mitigate the fault, and eventually repair the defective unit. Facilities to support error detection, termination, and recovery must be independent of system topology. The OS may provide dynamic topology and resource management facilities and enhanced device drivers for the graceful replacement of failed hardware, or the initialization of newly inserted devices. Many HA configurations depend upon the hot-swap of components, frequently referred to as ***Field-Replaceable Units (FRU)***. This functionality considerably increases up-time by allowing in-service replacement and reprovisioning of malfunctioning hardware without having to power-down, re-initialize or reboot the entire system.

Finally, the operating system needs to provide services that are specifically designed to provide autonomous fault-management capabilities (when appropriate), services to report faults externally, and control interfaces for the directed management of faults and resources from the middleware and application layers.

High availability designs impose unique and new requirements upon an operating system. To support high availability an operating system must have facilities to minimize downtime in light of underlying hardware that will fail. In a HA design, system devices and supporting software must be dynamic, and the topology and state of a system’s hardware and software needs to be maintained and reported to management software and system operators.

9.1 OS Robustness

Improving the reliability of the OS is the process of modifying existing software to improve its behavior in terms of reliability, availability and faults. This process consists of adding to the OS the ability to:

- Detect, diagnose, isolate, and recover from faults in both hardware and software
- Avoid accidental fault masking; if a fault occurs, ensure that it is appropriately reported and handled
- Provide warning of unusual conditions that may, when combined with other higher-level system knowledge, warn of an impending fault
- Provide information to assist isolation of the failure post-mortem
- Provide dynamic kernel and application event trace and profiling facilities

Often, the above capabilities include a higher degree of stabilizing the code and ensuring that the software conforms to appropriate and established software practices, such as code verification, code coverage analysis, elimination of dead code and consistent error code generation.

Other examples of enhanced OS capabilities that improve the reliability are discussed in the next several sections.

9.1.1 Error Code Checking

The commonly accepted software practice of always checking the returned function code for error indications applies to OS design as well as to application design. In a robust OS, all function calls should parse the return code for potential fault information. If detected, the calls should log the exception and take appropriate autonomous error handling (fault management), or if warranted, halt operation until the fault can be cleared by an external directed operation. This fault management mechanism within the OS layer is fundamental to improved reliability.

9.1.2 Hardened I/O

Hardened drivers are enhanced device drivers that do not assume that the devices they control are always reliable or available. I/O interfaces should be designed to expect hardware failure, and attempts to ensure that system functionality will not be significantly degraded or locked-up by the failure of an external device. All I/O processing should scrub latent faults with some form of limit checking for reasonable and expected values. Further, no I/O operation should wait indefinitely on a faulted component. The access operation should have a reasonable timeout, exit and recovery process on any pending request. The purpose of hardened drivers is to detect and isolate faults as much as possible. Driver exceptions and faults are typically logged, as well as reported to the middleware layer.

9.1.3 Argument Checking

Every data structure and function argument that can be rapidly checked for correctness should be checked. In the case of data structure corruption, regenerate the data structure, if possible. In the case of wild arguments, the OS should return a reasonable error status, and the error should be logged.

9.1.4 Consistent Programmatic Response

An OS with consistent and deterministic behavior can aid in the rapid detection of time-interval related faults, as the time relationships are known with greater certainty. This certainty allows fault detection and recovery mechanisms such as hardware and software failsafes (e.g., watchdog timers) to operate with a finer time resolution.

9.1.5 Avoidance of Arbitrary Limits

Enforcing unnecessary arbitrary limits on the length or number of any data structure, by allocating all data structures dynamically, can directly affect reliability over time. A HA system, designed for minimal downtime and in-place upgrades, may have peak resource usage that was never foreseen during design or test. The ability to dynamically extend OS structures eliminate this point of failure. However, if the dynamic allocation is not allowed by system design or fails due to resource limitations, the failure can be logged, the appropriate in-band error code communicated to the application and exception notification information can be used by the middleware or application to recover from this exception condition.

9.1.6 Appropriate `panic()` Behavior

A catastrophic system failure, or `panic()` routine is used when a failure occurs which cannot easily be recovered from. Frequently these failures are system data structure corruptions. The result of the `panic()` routine is to crash the system, with the obvious impact on availability.

By design, a HA-specific OS seeks to minimize panics by replacing most system exception conditions with appropriate autonomous fault management behavior. However, if the fault is such that correct operation of the subsystem is no longer ensured, an indication should be set so that further attempts to access the subsystem fail. If it really is the case that the entire OS's operation cannot continue, then an OS panic() should be the only acceptable action. This will force redundant OS components to assume the role of the failed system. Whenever possible the panic routine should capture as much descriptive information as possible, on the abnormal termination. It is highly desirable to support implementation-specific flexibility in how and where this information is captured.

9.1.7 Handling of Spurious Events

The OS should recognize and handle potentially spurious events, such as spurious IRQs and controller and bus glitches.

9.2 Notification

However fault domains are constructed, a critical hardware capability is the detection of failures of fault domains and communication of those failures. This may be communicated ***out-of-band*** through a management data channel, or ***in-band*** via unambiguous observable behavior (or non-behavior). A primary example of the latter is ***fail-safe*** behavior, where a fault domain contains a self-checking capability, which causes it to promptly shut down when a fault is detected. The resulting shut down is then observed by other parts of the system.

Beyond the immediate communication required for fault diagnosis and isolation, hardware fault domain failures must also be communicated to appropriate middleware (or people) in order to trigger recovery and repair actions.

When a system contains fault domains, which are effectively in a standby mode, there is a need for detection of latent faults in these domains. That is, if the primary failure detection mechanism is observation of normal operating behavior, the hardware may need to provide a separate mechanism for detection of faults in fault domains that are not normally operating.

9.3 HA-Enhanced OS Services

Many of the software mechanisms required to support fault management operate in the OS level because they are intimately related to kernel activities, or are provided at this level for performance reasons. OS services such as protected address spaces or process and data resiliency provide inherent mechanisms that aid in fault management. Other OS layer capabilities like a secure file system are relatively independent and usually require no ***directed intervention*** by middleware. An example would be a journaling file system that speeds system restart and file system checking. The middleware does not have to control these functions because they are included in the OS. Hardened device drivers that check for hardware errors may be required to report exception information directly to the middleware layer. Open-standard interfaces provide a well-documented and consistent set of interfaces for fault reporting, fault logging as well as mechanisms for OS layer capabilities to ***register*** with the middleware layer.

9.3.1 Memory Protection

Modern software divides computer memory up into regions, principally for program code and several types of data. Without a hardware ***Memory Management Unit (MMU)***, these divisions are soft, enforced only by the way that development tools lay out memory and by how programmers follow the layout discipline. Large multi-process or multi-threaded programs further divide these gross regions, giving each process, task, or thread of execution its own chunks of memory for code and data. Among the hardest bugs to find are those that violate this neat structure — when programs unintentionally modify their own code regions, accidentally corrupt data structures, or violate the code or data regions of other threads, or even the operating system kernel itself.

Hardware-based memory protection offers greater security and robust application code by isolating the various regions of a program and using the integrated hardware MMUs on modern microprocessors. The MMU erects walls around code and data by defining memory segments for each and restricting access to those segments: code segments are read-only, disallowing accidental self-modification. Data segment access is restricted to the current executing process or via read-only data, and may be protected in the same manner as code. Thread stacks can be laid out with virtual comb-tooth address gaps between them, preventing inter-stack corruption through ***walk-over.***

When a program attempts to write over itself or violates data access restrictions, an exception occurs, the executing program is interrupted, and the offending program location can be easily located and repaired. In multi-process HA systems, one errant thread will be stopped from corrupting the entire application, and may be restarted or replaced without interrupting mission-critical operation.

Virtual Address Spaces

An operating system can accomplish this leveling process by assigning each program or process its own MMU-enforced virtual address space. Then, each process can treat the entire computer address space that it can see as its own. With this type of memory protection the process does not risk corrupting the memory space of other processes in the system. The operating system maps all program memory accesses, from the program's vision of memory (logical view) to the available collection of system memory resources (physical view) through the MMU. This mapping is completely transparent to the process and its programmer, focusing programming effort on algorithm design, not implementation particulars.

9.3.2 Process Handling Overview

In a Linux/UNIX process style model, programs are executed as processes. Each process has its own protected address space so that an access to unallocated or illegal memory is detected immediately, not allowed to corrupt another process, and is trapped. This protection provides fault isolation for a wide range of software faults.

Resource Recovery and Containment

The process model also keeps track of what resources are allocated on behalf of a process. When a process or thread opens a file, allocates memory, or attaches to shared memory, resources are allocated by the OS and then returned to the system when the process exits. A process can create other processes that become child processes. The parent process sets limits on the amount of resources that a child process can allocate. When a child process exits, its parent can get status information, including why the child process exited.

Processes can also be forced to exit asynchronously by sending a process kill signal, and can aid the fault diagnosis process. The ability to recover resources from a faulted, or shed, process aids in fault recovery. The open-standard ***Process ID*** (***PID***) structures allow the middleware to easily determine what applications are currently loaded and running.

Dynamic Loading

Dynamic processes can be loaded, executed and unloaded without restarting the system or OS. This allows the system to be dynamically reconfigured or upgraded without having to reboot. Moreover, since processes are completely contained within virtual address spaces, restarting a single process after a failure is a safe option and obviates the need to restart the entire system. Such dynamic restart capabilities at the application and driver levels provide a foundation for high availability and allow hot upgrades of executing software as well as device drivers and their related hardware. The ability to dynamically create new processes and load programs may also be used for application rollback, under the directed control of the middleware. This capability is also useful in clearing transient software faults.

9.3.3 I/O Device Drivers

The OS layer provides the access to the input/output system. I/O requests are passed to device drivers that service that type of request. The device drivers handle the hardware specific aspects of I/O so that applications can be hardware independent. Because all I/O access goes through the kernel, I/O requests can be redirected should the system be dynamically reconfigured. This feature can be used to mask I/O failures at the application level as long as there it is a virtual device, or to implement some other form of hardware redundancy to take over the I/O operation.

Dynamic Device Drivers

An OS capability that allows devices drivers to be dynamically loaded supports the in-place provisioning of new hardware without re-starting the OS.

9.3.4 Signal IPC Mechanism

An OS that supports signals can use this capability for asynchronous notification and control of threads. Signals can be used to start, stop, or force the exit of threads or process.

In the case of CPU fault, a signal sent to the thread running the code that caused the fault provides a dependable method for logging and reporting the fault.

The signal timers can be used on a thread basis to provide a software watchdog function. The timers can also be used to generate a periodic signal to remind the process to use the middleware checkpoint capabilities.

9.3.5 Management Access to Kernel Information

Structures and Process States

Accepted industry standard information blocks (such as the Process ID table, memory in use, disk free blocks, CPU idle time, etc.), provide a mechanism for external access and visibility into the OS layer. Access to system information through standard calling mechanisms provides a well-understood programmatic access to OS layer information. Both the application and middleware layers frequently use this access interface to the OS.

OSs may also support a structured and polled ***system MIB***. This MIB typically structures the kernel information according to a published structure. This information can be directly incorporated into the element management mechanism, or parsed, ad hoc by the middleware, to garner required system status and state information. This management interface to the OS layer typically conforms to one or more of the industry-standard network, element, or web management protocols.

A desirable capability at the OS layer is the autonomous generation and communication of kernel information when certain installable characteristics are approached, matched, or exceeded. This OS layer management capability is often used to provide immediate and proactive information of unusual conditions that may, when combined with other higher-level system knowledge, be used to predict impending faults and to take corrective action in advance of actual failures.

Structured Error Logging

The purpose of the event logging function is to provide a mechanism for logging system, application and exception information and for subsequent processing of that information. The active system log should itself be highly available to accept event records. There are typically extended functions to archive and distribute the system event log. Fault logs can be invaluable when trying to repair a system. Since faults due to configuration errors may occur at boot time, the fault logging facility is available soon after the initialization of the kernel.

If an abnormal condition is detected upon checking a return code, argument, or data structure, the condition must be reported to the user and recorded in the system log. This is true even if an error is recoverable, as an HA system must not hide errors from the prediction and detection devices.

Desirable OS layer enhancements to the system log function would continually parse the entries by type, frequency and severity and could provide asynchronous notification of exceptions to the middleware layer. Ultimately, the OS should provide both asynchronous and directed access to this information.

Crash Dump Information

In the unavoidable event of a system failure (e.g., OS panic) or application fault (e.g., core dump), as much information on the error as possible should be captured, reported, and saved (if only locally) for post-mortem fault debug and diagnostics. To provide more rapid debugging, it may be desirable to have the option of limiting information dumping during debug.

9.3.6 Configurable Restart/Reboot Behavior

Mechanisms to control the OS behavior upon reboot (typically after a crash) or restart are valuable in a HA system. Being able to specify that a faulted system should not automatically reboot after a failure may allow analysis of the failure and potential diagnosis of the exception condition. It also avoids potential introduction of an unstable component into the overall system. After an in-place upgrade or a fault, it should be possible to do either a fast restart, a rollback, or a failsafe restart.

9.4 Hot-Swap Software Requirements

Several key software features facilitate hardware hot-swap. Hot-swap for HA requires that system software resources allotted to a board be reclaimed when the board is extracted, and added when a new board is inserted. It is required that device control software be installed dynamically and linked to a running operating system upon hardware insertion, and that applications can be quiesced (or at least notified) such that pending communication/operations with the board be halted or suspended and redirected upon removal. Explicit control of device drivers and their association with hardware is required so systems integrators can create and enforce policies governing the working of the system.

In a traditional system the hardware, control software (typically a device driver) is very tightly associated with the hardware it controls. Understanding of (static) address and interrupt mapping from a processor to a device most often has been the responsibility of the device driver writer.

9.4.1 Device Resources Abstraction

It is easier to support the hot-swap of I/O cards if device driver design can be uncoupled from the topology of the system. A hot-swap aware driver can be simplified through isolation from address and interrupt mapping and ***resource allocation***. The OS may provide a ***device-resource-facility*** between the device driver and the mechanisms for finding the device. This can be used to abstract the specifics of device address and interrupt mapping, effectively removing them from the driver and segregating them to a driver support system with its own API. Moreover, this device-resource-facility can initialize the bridge chips and set some initial policies governing expansion areas on the additional buses connected by the bridge chips.

9.4.2 Stateful Device Driver Model

In a system that does not support hot-swap or high availability, applications programs and systems code must assume that hardware resources (devices) and the associated drivers are always in a ready state. Adding the dynamic nature of hot-swap devices to a system not only adds complexity to system software, but invalidates such static configuration assumptions at all levels. As good device driver style dictates, such complexity is best isolated in the device driver and specific interfaces, since many aspects of hot-swap functionality are device specific.

A driver model based upon open standards, like Linux/UNIX, provides a set of standard entry points that are used to interact with a device at an application level.

A stateful device driver model not only conforms to this standard style I/O interface, but also provides control interfaces to support the dynamic insertion and removal of devices through the use of a role state machine. By using a state machine, the driver can be structured to induce the needed state transition through the use of a recognized mechanism such as `ioctl()` (I/O control) commands. This driver model allows some level of diagnostics to be run on the specific device while the system continues to provide service.

In order to allow off-line testing of a device while the operating system remains in service. It is necessary to have at least one device state, a ***diagnostic state***, that allows for continued processing of the management control interface (ioctl) while keeping the user access to the device stream in a quiescent state.

9.5 Support of the Application Layer

The process model can be used to extend and control depend-upon relationships that may define simple functional groupings and software fault domains.

10.0 Layer-Specific Capabilities – Management Middleware

In availability management, hardware or software faults are not avoided, but are expected to occur and the system is designed to anticipate and work-around faults before they become system failures. Thus, instead of counting on the hardware to avoid faults, availability—and especially high availability— design relies heavily on management software to mask and manage faults that are expected to occur. Fault management takes practical precedence over designing for fault avoidance; the goal is to anticipate faults and execute fault recovery as quickly as possible.

High availability depends on each part of a system working reliably and in concert to deliver services without interruption. Hardware and software redundancy enable management software to replace failed components with the appropriate standby components so that services can remain available; accomplishing this with minimal downtime or loss of client state requires a unified approach to collecting, analyzing and acting upon system state information.

HA also requires a high-performance solution. Given both the architectural complexity of service availability systems as well as the performance standards required to actually achieve continuous availability, it is crucial that the management software itself carries a low overhead. The management software must be optimized for speed, efficiency and footprint, so as not to impede performance and undermine overall availability. Additionally, to account for the fact that the management middleware itself is subject to failure, the management middleware should have the capability to monitor its own health as well as the health of management middleware on other nodes within the system.

The entire availability management cycle must operate automatically in real time, without need of human intervention. Information about the system must be collected and assessed so the system can be managed. System components must be represented and their status, topology and dependencies must be modeled and monitored. System anomalies or faults must be quickly detected and diagnosed. Fault data must be provided to an intelligent availability management service so that it can quickly and appropriately respond by initiating actions that reconfigure the status and functioning of the components as needed to maintain service. In other words, the system must be self-managing and self-reliant.

Thus, implementation of service availability requires management software that can do all or at least some of the following:

- Collect system data in real time
- Configure and maintain state-aware model of the total system
- Checkpoint data to redundant components
- Detect, diagnose and isolate faults
- Perform rapid, policy-based recovery
- Dynamically manage configuration and dependencies of the components
- Provide administrative access and control

These requirements are described in greater detail below.

10.1 Collect System Data in Real Time

All critical system components must be continuously monitored and managed in a unified solution. This includes hardware, software, operating system and applications. Availability management software, therefore, must include an interface to these components. A flexible approach to use for this purpose is an object-oriented framework using ***managed objects.***

A managed object is simply a logical representation of a physical or software component. Creating a managed object includes assigning its name and location, enumerating its relevant attributes, and defining the methods by which it is to be managed. Based on the defined attributes and methods, the management middleware can retrieve the appropriate current state information, populate and dynamically update management database tables, and process the information to determine if any action is required.

The following components are representative of the types of physical system components that can be represented by managed objects within management middleware:

Hardware:

- CPU cards
- Line cards
- Fans
- Power supplies
- Temperature/current censors
- Storage devices
- Alarm displays
- Communication links

Software:

- Operating system (and sub-components within)
- Protocol stacks
- Applications

When one of these components is represented by a managed object it is referred to as a ***managed component.***

The management middleware should also handle the notion of ***logical system components*** in which a group of redundant components act together to provide a single service within the system. For example, a group of three redundant fans may have a logical managed component to represent the group.

In addition to collecting data, managed objects provide the ability to control a given component via methods. Operations include auto-discovery, event handling, and various control functions to configure or otherwise operate the component. These operations are used by automated service availability management as well as by external management interfaces such as a web-based user interface or an external SNMP console.

10.2 Configuring and Maintaining State-Aware Model of the Total System

Availability management requires a system-wide model that can represent all managed components in the system, changing information and the intricacies of each component's dependencies and interdependencies. The management software needs this information to make quick and appropriate reconfigurations when necessary.

Components that rely upon another component or set of components in order to work can be represented not only as a parent/child dependency relationship, but more importantly as a dual-connected directed graph (***digraph***) in which each component is linked to every other component upon which it depends, and each component is linked to every component that depends on it. Such dependencies can span a complex array of branching interdependencies. An example of a system and the directed graph which represent it are shown in Figure 16 and Figure 17. Availability management must understand role relationships and the consequences of component failure and role reassignments. Both the dependent and the depended-upon relations need to be monitored in a system model and dynamically reconfigured as needed.

Figure 16. Example of a Managed System

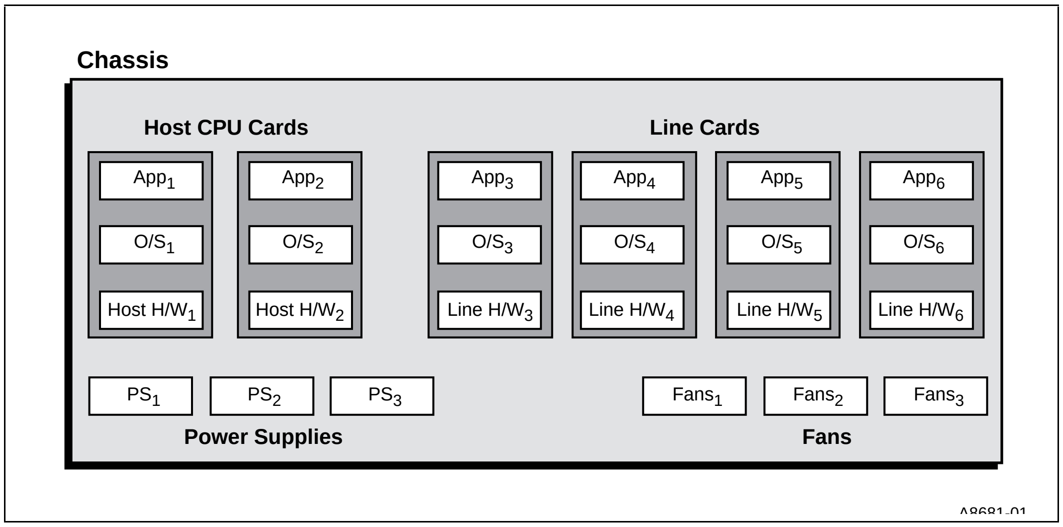
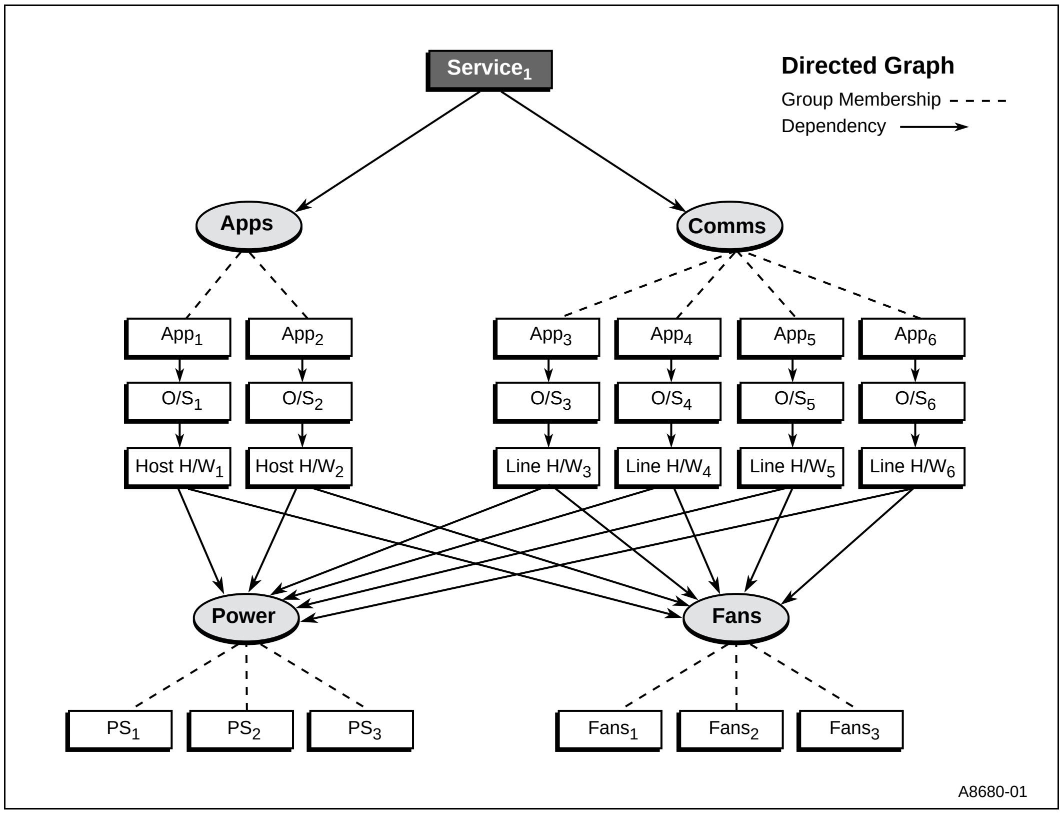


Figure 17. Example Directed Graph Describing a Managed System



In addition, service group dependencies also must be modeled, monitored and managed. Service dependency groups are collections of managed components orchestrated to cooperate in redundancy configurations in order to secure the availability of the services the system provides. These sometimes-complex dependencies and interdependencies not only overlap architectural layers, but they often involve more than one node and include heterogeneous components and platforms.

The topology and data represented in the model must be stored in a high-performance database in order to provide current state data that can be rapidly updated to reflect ongoing changes of state in the managed components. In addition, to secure the availability of the management service itself, the database should ideally be replicated to a redundant node in case the component hosting the primary management database should fail.

Managing component faults and failures requires rapid access to the schema of the physical (parent/child) dependencies that enable the system to function. Understanding and accessing the knowledge in this system model quickly is crucial to effective service availability management.

10.3 Checkpointing Data to Redundant Components

Redundancy is key to availability. In many cases, components are stateless and can be replaced on-the-fly by a redundant standby component without interrupting service. However, in some cases the component contains state information that must be preserved to avoid interrupting service during switchover to a standby component.

To provide service availability, the current state of transactions often must be maintained in a hot standby redundant component. This means that ongoing transaction data and application state data must be continuously delivered (***checkpointed***) to a hot standby location. Real-time checkpointing that provides transaction state data for graceful switchover requires a high-speed data storage and retrieval system and a fully-optimized method of messaging and communication.

To ensure that the heartbeats and checkpointed information are properly received by the standby component, it is recommended that the paths used for this communication be redundant. This minimizes the impact of a failed communication path due to hardware (i.e., failed interconnect) or software (i.e., failed TCP/IP stack).

10.4 Detecting, Diagnosing and Isolating Faults

At the most basic level, ***fault management*** must detect faults and initiate notification and an appropriate recovery action. The fault management required to achieve service availability is necessarily more complex. It must detect not only active faults, but also latent faults, in order to anticipate and avoid critical system errors. It must perform sophisticated diagnosis and root cause analysis for quick and appropriate containment of faults with minimum system impact. And it must intelligently determine and initiate policy-based recovery actions.

Detection is the discovery of a fault or symptom. It is the identification of an undesirable condition that may lead to the loss of service from the system or device. Detection may be based on error detection (through direct observation or correlation of multiple events in location or time) or inference (by observing other behavior of the system). Detection is accomplished by creating fault detectors that are associated with data collectors in managed components.

The management middleware should provide a variety of methods for detecting faults. It should actively monitor the system model and its managed components for state changes that will impact the availability of a service. It should also monitor the system components directly via methods contained in the managed components to receive fault events that are sent directly from the component. Ideally, the management middleware should also provide the ability to analyze trends in system performance as well as transient and recoverable events in order to predict faults.

Diagnosis analyzes one or more events and system parameters to determine the nature and location of a fault reported by a detector. Identifying the originating fault (root cause analysis) is part of diagnosis; this can be difficult, since errors tend to multiply quickly and a single fault may lead to multiple errors. In complex systems, both diagnosis (including root cause analysis) and the subsequent isolation of the fault depend on information about the current configuration topology and dependency relationships represented in the system model.

Typically, a fault diagnosis may automatically trigger local actions within the component, forward data to management middleware for action, and also may generate notification of administrative personnel. The management middleware should handle each of these cases.

Isolation contains a fault to keep it from spreading throughout a system. Configuration and dependency relationships must be understood in order to partition appropriate fault containment regions.

10.5 Performing Rapid Recovery

In general, completion of the fault management cycle includes recovery from the fault, as well as reporting to administration and repair and reintegration as needed. In complex systems with both parent/child dependencies and multi-layered service dependencies, the management of fault recovery actions requires multiple factoring of hierarchies of cluster-wide dependency and availability issues. It needs to be handled by the higher-level availability management functions. In order to build a high availability system, redundancy is often configured across multiple nodes. Fault management on a single node can handle local recovery, but the cluster-unified management is required for fault management that crosses node boundaries. Notification and repair are part of interfacing outside the self-management structure.

Recovery includes any actions taken to restore the system to service from fault or failure. These actions can cover a wide range of activities from restart of a failed application to failover to a standby hardware card. The recovery process is often multi-step in that several actions must be taken in a prescribed order. In some cases, the recovery process is multi-tiered in that, if a particular action does not recover the system, some alternative action must be taken. The management middleware should contain the knowledge of the appropriate recovery action for the failure of each managed component in the system, whether it involves restoring that component to full operation or switching over to a redundant component. While not technically part of the recovery process, similar recovery actions may be taken to proactively detect and resolve latent faults.

Reporting is the notification and logging to systems or people of the diagnosis made as well as any actions that were taken automatically. For example, if an application crashes, it might be both recovered by restarting via pre-set protocols and then reported to a system administrator via an e-mail or page. Notification to the outside world that an event has taken place is the first step in the repair process. Management middleware should provide a variety of notification methods via e-mail, SNMP traps, event logs or other messaging paths.

Repair involves the repair or replacement of hardware and software components as necessary. ***Reintegration*** returns the working component into the system. Assuming a system is designed with redundancy, the functions of a failed or degraded component are switched over to the standby component as a recovery action. Once the failed component is inactive, repair can begin. Software repair could involve patches or installation of an upgraded version of the software; either can be accomplished automatically via remote upgrading. Hardware components would be decommissioned automatically, but are usually repaired or replaced manually. Reintegration support by availability management—such as automatic detection and role assignment—avoids downtime or loss of service.

Quick and accurate detection, diagnosis and isolation of faults (and symptoms that could lead to faults) enables fault management to prescribe and initiate appropriate recovery actions. Recovery actions must take into account the dependency consequences of any reconfiguration.

10.6 Dynamically Managing Configuration and Dependencies of All Components

The management middleware should provide cluster-aware availability management that is responsible for initiating the actions and orchestrating the role assignments that maintain service availability. This function should not only register and track component membership in the system, but it should also understand and manage the overall system dependency and redundancy configurations. It thus relies on a dynamically populated system model.

The management middleware also controls service availability groups (the collections of managed components in redundancy relationships such as 2N or N+1) and makes role assignments (active, standby, spare) within them.

10.7 Providing Administrative Access and Control

The availability functions of the management middleware described so far comprise an automatic, self-managing system. The administrative functions described here provide the necessary link to human and enterprise management system intervention and control. The goal of high availability is greatly supported by simplified management, including a single point of access to a unified, system-wide representation.

Effective external administration requires local and remote access to the managed components—a central location for configuration and management of an entire cluster, the service availability groups, and individual managed components. Administrators must be able to monitor the status of managed components, configure the components, and receive alerts when errors occur in the cluster. Administrators need both system-wide views and drill-downs to individual components.

Management middleware must provide a set of services for the ongoing maintenance and management of the system. Outages due to planned system maintenance or hardware or software upgrades are not acceptable in many high availability system designs. Among the important functional requirements are remote, automatic upgrading; retrieval of components data; provisioning and deploying additional nodes; integration into CIM management structures, notification of alert conditions, and integration with SNMP network management systems.

10.8 Providing a Consistent Interface to Applications

In many HA system designs, application developers must incorporate capabilities into the application to interface with the management middleware. There is a strong desire to keep interfaces consistent so that applications are portable across various implementations of platforms, operating systems and management middleware. These interfaces are discussed in detail in Section 5.4.

10.9 Alignment with Standards

Much work has been done throughout the industry to develop standards in the area of system communications and management. Where possible, the management middleware should embrace these standards. Among other things, this enables system manufacturers to preserve previously-developed or newly-designed capabilities within other parts the system. Examples of relevant standards include:

- CIM – Common information model
- CORBA – Common Object Request Broker Architecture
- CMIP – Common Management Information Protocol
- SNMP – Simple Network Management protocol
- X.731 – ITU recommendation State Management

References for these standards can be found in the bibliography.

11.0 Layer-Specific Capabilities – Applications

When reviewing capabilities for high availability systems from an application perspective it is important to understand the application objectives. Many different applications are required to make a system. There are many ways to classify applications and the capabilities of applications to participate, control, or operate within a highly available system. Since the goal of most applications is not specifically focussed on fault management, the system should augment that task by providing a management interface to allow an application to monitor and control operations. Additionally, an application should be able to supply its health information to a management tracking function.

To that end, the following groups can be defined for application capabilities:

- Status
- Notification
- Failover
- Recovery
- Resilience

11.1 Status

An application needs to know enough about the system configuration to determine if it is capable of running. For example, it must know if the hardware, storage, and/or communications path is in existence and operating. This requires that the application be able to extract the knowledge of the system through a process called ***discovery***. This information can be very platform and software specific, but general concepts can still be applied. Simple management interfaces already exist and can be leveraged to help with discovery. Internal standards such as CIM and IPMI can be used, as well as network-based management interfaces such as SNMP, to provide some or all of the information. A practical view of recovering status would be to try to have the widest breadth of information and a limited depth. This could be provided in a common way on all platforms and operating environments. If the application needs additional hardware specific information, then it should be able to get that using a platform/vendor/operating system specific method.

An additional area of concentration should be software configuration. This is the ability to survey or inventory the software configuration of the system. The information that is needed includes service, library, file, and version information.

Finally, the interdependency of processes, services and daemons that can be used by some applications needs to be determined. Items like pipelines, shared memories, and protocol stacks can all be shared between applications, hence complicating the dependency tree.

11.2 Notification

Applications need access to timely notification of desired events. These events could be from hardware or other components in the system or events in the form of a heartbeat from a mated application running elsewhere. Again, a common method of hooking in for an asynchronous operation that covers a reasonably wide level of functions should be through a common interface. Then, additional detailed information can be extracted in specific ways. The event notification should allow for some level of filtering. Common methods would be to use a publish/subscribe to register the application for others to see and to be managed.

11.3 State Preservation

A common method for state preservation should be provided for an application. This preservation should allow for an application to restart at a known state. This preservation may require some level of replication of the data. In the type where an application is to restart on the current processor, the preservation could be in volatile, non-volatile memories or even on a storage media like disk or tape. For an application that is started or running on a standby processor, the state preservation data would be transported between the active and standby processor. This is referred to as ***checkpointing***. The underlying mechanism to provide this service needs to be able to assure that the information is as supplied by the source (data integrity), as well as timely. Additional items to consider with checkpointing include the idea of a suite of programs. If there is more than one program, items like synchronization, sequencing, and data committing need to be considered. Checkpointing should also include a mechanism that identifies whether the application is providing the service, sharing the service, or backing up the service with another process.

11.4 Recovery

An application's capabilities to recover from faults in the system are dependent on the capabilities of the system. From the application's perspective, the underlying system should provide basic tools to manage resources. It should not waste or lose resources. The objective of the recovery process is to restore the system to an operating state, even if it is in a reduced capacity.

An application in a highly-available system will need to be able to control the flow of data, be able to start other processes or stop other processes, replace or upgrade itself, restart, and even reboot the underlying system.

One capability of a running operating system is to prevent it from getting congested. Flow control prevents a system from failing at the application level by consuming all the resources or by blocking I/O activity. The application needs to use non-blocking I/O or timeouts on I/O processing to prevent this from occurring.

In some application environments, the ability to start or stop processes from an application (***process control***) needs to be supported. A primary focus on this would be the ability to launch diagnostics in the event that some component was removed from operation due to a fault.

Applications need to keep track of the versions of their content. For example, versions of files and libraries used to build an application should be kept. Based on this information a recovery operation can replace the entire application or just specific pieces of it.

Restarting an application can be an important step in a recovery of a system service. The faster the restart of the application the sooner the service is restored. If the setup and processing of an application are stateless or require minimal state information, then a cold restart can be performed. This can be either on the current processor or on a redundant processor. If a larger amount of state information is part of the operation or processing of the application, the warm restart of the application is necessary. The interface to the checkpointed information is then need to reactivate the application. In the event a redundant process is used for either the recovery of an application or the upgrade of an application, careful consideration is needed to address the use of resources such as file locks, semaphores, or device handles. Understanding the specific information about the device handle, semaphore or memory resource and how it will be accommodated on the receiving side is critical to a rapid restart.

It is inevitable that a situation will occur which requires a rejuvenation of the system. This reboot operation is needed from an application trigger to be used as a last resort in part of a recovery operation.

11.5 Resilience

Resilience is the property of a component that allows it to continue full or partial function after some or multiple faults occur. Thus, resilient components provide a higher availability level than comparable non-resilient components.

The basic category of resilience applies to the actions taken from an operation. Beyond style, platform features for common operations should be provided. All interactions with the high availability interfaces need to check the inputs and validate the outputs. In addition, common techniques for redundancy and voting operations can be provided to improve the resilience of an application.

Properties of resilient components include the ability to detect a fault and compensate for the fault. This may be utilizing alternate resources, or correcting the error. For example:

- Error-correcting memory can be an example of a resilient component. If errors are detected and corrected, an area of memory can be taken out of service and moved to another section of memory that is operating correctly. Then the memory addresses would be re-mapped to use the good memory.
- In software, database data is required to be correct, but disk writes of data with interdependencies cannot be written atomically. Databases are resilient to non-atomic writes, by adhering to a strict sequence of writes so that any errors can be corrected.

Using these and other similar methods of design can make an application or system resilient to faults. If one analyzes the faults that could occur in a system and puts methods in place to automatically resolve the faults, it increases system availability. The highest service availability values are usually reached with a combination or resilience, redundancy, and good design practice.

12.0 Glossary

5-nines – Maintaining availability 99.999% of the time.

2-N – A method of redundancy where there is one component in standby for every component in operation.

Active – Component currently in use providing a service.

Active Fault – A fault that is currently causing an error. Active faults are not necessarily detected, although they should be in a well-designed HA system.

Active/passive – A system where one redundant component is Active while the other is available, but not in standby mode.

Active/standby – A system where one redundant component is Active and the other is in standby, ready to take over with little switchover time or data loss.

Administrative Notification – Notification that is sent though other than the management control system. This includes notifications with alarms, lights, separate display panels, etc.

Application software – Software which runs above the OS and either above or next to management middleware. This software typically performs the main functions of the computing unit, or provides direct support for these functions.

Atomic component – A component that, for the purpose of system description, is not decomposed into other components.

Attributes – In object-oriented system design, attributes are the characteristics (or parameters) of an object. See also Methods.

Autonomous Notification – Notification of an error by a unit for an error within that unit. For example, an OS stack overflow which is detected by the OS.

Checkpoint – The process of copying data from a unit to memory or to the unit’s standby to allow for switchover while maintaining state.

CM – See Configuration Management.

CMIP – Common Management Information Protocol.

CMIS – Common Management Information Services.

Cold Restart – Restarting a device from power off or with no state information.

Commercial Off The Shelf (COTS) – Products which can readily be purchased by anyone.

Confidentiality – The attribute of not passing information to a unit which does not have rights to it.

Configuration Management – Controlling the configuration of a system by setting which components are active, standby or spare and by monitoring the status of components to determine if they are available, running or failed. Additional configuration management features can provide more granularity on the control and status of components.

Corrective maintenance – Maintenance for the purpose of fixing a known or expected to occur error in the system.

Curative maintenance – Maintenance for the purpose of fixing a known error in the system.

Data Isolation – Using memory management to keep data from one program, task, or thread from interfering with data from other program areas.

Degraded condition – A condition in which a system is still operating and providing service, but perhaps not as rapidly or without an available standby unit.

Dependability – The attribute of a system such that one can rely on its responses. Dependability is a combination of Availability, Reliability and Integrity.

Detection – Finding a fault in a system.

Detection Frequency – How often a system or signal is sampled to verify that it is not in an error condition.

Deterministic – The attribute of a system such that one can determine when events are going to occur.

Diagnosis – Determining which component caused a fault in a system so that recovery can begin on that component.

Digraph – A Directed Graph which shows components and their dependencies.

Direct Detection – Detection of a fault by directly observing it. For example, detecting high CPU temperature by measuring the CPU temperature instead of the outlet air temperature.

Directed – Directed activities are those that are controlled by a component other than the one performing that activity.

Discovery – The process of determining what devices are in a system.

DMR – See Dual Modular Redundant.

DMTF– Distributed Management Task Force. A standards body working in the area of system management. The DMTF is responsible for CIM and WBEM.

Download – To load new software onto a system.

Dual modular redundant – A redundant system in which there are two modules operating in parallel. If they do not give the same information an error is detected and a recovery needs to be made.

Dynamic reconfiguration – The ability to change a system configuration while the system is in operation.

Error – The occurrence of a component not providing the correct information at the correct time.

Fail-safe – When an error occurs the component will fail in a way that indicates to the rest of the system that it is no longer reliable. In many cases that means that it will just shut down, rather than chance giving out incorrect data.

Failure – A system fails to provide service at the level it was designed to provide.

Fault – A problem in a component where the response was either not correct or not timely.

Fault detector – A hardware or software component that checks for faults.

Fault domain – A group of components that is replaced when a fault is detected in any of the components.

Fault management – The process of Detection, Diagnosis, Isolation Recovery and Repair of a faulted component. Fault management works in conjunction with configuration management to change redundant components.

Fault prediction – Using information gathered by a management system to predict when a fault may occur. If possible, preventative maintenance can then be performed to keep the fault from occurring. Fault prediction relies on parameters such as time in use, temperature and error rate.

Fault prevention – Using best known methods during the design phase of a system to prevent faults from occurring when it is deployed.

Fault removal – Removing faults during the design phase of a system by validation/verification, diagnosis and correction.

Fault tolerance – The system attribute of being able to operate correctly while faults are occurring.

Fencing – Removing device entries in an I/O subsystem so that the component no longer receives inputs and can no longer change outputs.

Field replaceable unit (FRU) – A hardware component that can be replaced in a repair process. FRUs are typically boards or modules that can be easily swapped out in the field.

FM – see Fault Management.

FRU – see Field Replaceable Unit.

HA Forum – An industry group with the goal of promoting open standards for high availability systems. The HA Forum generated this document.

Hardened driver – A software driver that has been written so that it will not lock-up or return faulty data to the OS, no matter what its associated hardware does. Hardened drivers are a key part of an HA system, and must be written to make use of HA features within the hardware and the OS.

Hardware platform – The hardware and firmware upon which an OS, middleware and applications are run. The hardware platform typically includes BIOS and diagnostic software.

Heartbeating – Sending a periodic signal from one component to another to show that the sending unit is still functioning correctly.

Hot restart – Restarting a component while the system is still operational or partially operational, typically with the new component picking up state information from memory rather than doing a complete initialization.

Hot-Swap – Changing a board or other hardware component in a system without shutting the system down.

In-band communication or message– Transferring information over the primary communications channel or bus. The primary channel is the communications framework, protocol(s) and hardware used for the majority of inter-process communications.

Indirect Detection – Detection of a fault by a method other than directly measuring or comparing the value which is faulted. Indirect detection is used for time-based errors and where direct measurement is difficult. For example, chassis temperature can be used to indirectly detect fan speed or CPU temperature problems.

Indirect Notification – A notification of a fault in a component which is initiated by a second component, typically based on a time-out or similar indirect condition. See Indirect detection.

In-line Notification – Notification of a component fault which is communicated directly to the functional block next in line to send or receive data to/from the faulted component. This is different than ***in-band*** communications, defined above.

Integrity – The attribute associated with a system which always returns the correct response or no response.

Intelligent Platform Management Interface (IPMI) – A standard which specifies a protocol and methodology for monitoring and controlling a hardware platform with or without an OS running.

IPMI - see Intelligent Platform Management Interface**.**

Isolation – Protecting the rest of the system from the fault by disconnecting the faulty component (either physically or logically) and perhaps substituting a safe output value. If a safe output value is substituted the system must be made aware of the failure, otherwise the system is not fail-safe.

Latent Fault – A fault that has not yet caused and error or been detected. A Latent fault is created by an incorrect situation which would cause a fault under certain conditions. The fault is latent if it has not yet been found because the conditions needed to cause it have not yet occurred. See also Active fault.

Logical Isolation – Using software methods to isolate a component from the rest of the system. This typically involves removing the component from I/O and process tables, but may also require re-routing around the component by changing component addresses.

Logical system component – A group of components that acts as a single redundant component to provide a service.

MAC – see Media access controller.

Maintainability – The attribute associated with a system which can easily be upgraded and repaired.

Managed component – A representation of component in an HA system that can be individually controlled and monitored.

Managed object – A representation of a component in a system that shows that component's attributes, methods and dependencies.

Management information base (MIB)– A data structure that holds information on how a system is configured or functioning. These data structures are stored in memory and transferred to other management systems as a method of communication system status.

Management Interfaces – Interfaces in a system between the components and the management middleware.

Management middleware – The software within a system responsible for managing that system. This software is typically provided as a separate package, although some operating systems may also provide these functions.

Mean time to failure (MTTF) – The average time between one failure and the next one as measured (or projected) over a large number of failures.

Mean time to repair (MTTR) – The average time it takes to repair a component or system as measured (or projected) over a large number of repairs.

Media access controller – The device within a network controller just above physical layer which provides the physical address and access control.

Memory management unit (MMU)– A hardware unit on a CPU which translates virtual addresses to actual physical addresses in a system. It allows multiple programs to be written assuming the same address space, and then assigns that physical address space upon running that program. A MMU also prevents one program from over-writing another.

Methods – In object-oriented system design, methods are the functions that an object can perform. See also Attributes.

MIB – see Management Information Base.

MMU – see Memory Management Unit.

MTTF – see Mean Time To Failure.

MTTR – see Mean Time To Repair.

N+M redundancy – A system that has N components working in normal operation and has M components in standby. This is frequently reduced to N+1, where there is one standby component for every N operational components.

Node – Component of a system which itself is a complete computer system, containing hardware, operating system, and communications capabilities.

OAM&P – Operation, Administration, Management and Provisioning; the standard set of functions needed to control a system.

Off-line Diagnostics – Diagnostics which are run on a component with either that component or the entire system not operational. Off-line diagnostics are typically run to verify that a component is working correctly before putting it into service. They are also run to determine the detailed cause of a failure.

OMG – Object Management Group, a standards body working with object oriented design of computer systems. The OMG is responsible for CORBA.

Open architecture – An architecture in which the interfaces are well defined and available for anyone to use. Open architectures promote diversity and best-of-breed solutions due to competition, they also give engineers the ability to focus on solving problems instead of porting code.

Out-of-band communication or message – Transferring information over a special communications channel or bus, usually reserved for control or management purposes.

Physical Isolation – Isolating a component from the rest of the system by electrical disconnection, either using switches or by removing a board.

Platform management – Managing a hardware platform using control features of that platform. IPMI is frequently used to provide a platform management function.

Preventative maintenance – Maintenance performed on a system to prevent it from failing while it is needed in operation. Preventive maintenance can be either scheduled or triggered by fault prediction.

Process control – The ability for an application or other non-OS software component to be able to start and stop processes.

Process ID table – The list maintained by the OS of which process and threads are running. It typically includes information on the resources (memory, I/O and CPU time) being used and the amount of time that the process has been running.

Publish/subscribe – A communications method whereby an software component "publishes" a list of services it can provide. Other components can then "subscribe" to these services (or "register" with them) and get notifications.

Rebalance/Re-route – The process of changing which components are receiving and processing which messages. This allows processing to be moved from one system to another should one become over- or under utilized or should one system or component fail.

Reboot – To re-start a computer. **Cold reboot** occurs when the system is powered-off to reboot. **Warm reboot** occurs when the hardware is left running, but the OS is re-started.

Recovery – The system is adjusted or re-started so it functions properly.

Redundant – Having a copy of a component that can be used if the original component fails.

Register – To sign up with a software component to participate with it. See also Publish/subscribe

Reintegration – To take a component that had been out of service and place it back into a system in either the unassigned, standby or active modes.

Reliability – The attribute associated with systems that do not fail.

Repair - A faulty system component is replaced.

Reporting – The process of passing information on faults and configuration changes to the management middleware and other components that need to have it.

Residual signature – A software signature which is placed in code to show which version(s) of patches have been applied to a piece of software. The signature may be removable, but not without removing the patch it refers to.

Resilience – The property of a component which allows it to function after incurring a fault. Resilient components provide higher availability than comparable non-resilient components.

Resource allocation – The process of assigning I/O addresses, memory blocks and interrupts to a software component.

Remote Monitor (RMON) - A standard for remotely monitoring a computing system.

Robustness – The property of a software component, particularly an OS, that incorporates tests for many error conditions and has been designed in a way which protects it from errant behavior.

Role – The function a component plays in a redundant system. Typical roles are active, standby, and unassigned or spare.

Rolling upgrade – The process of upgrading software by making changes on one redundant component at a time, so that service availability is maintained

Safety – The attribute associated with systems that can not damage either the environment or the users.

Security – The attribute associated with systems that do not allow information to flow to unauthorized components or users.

Service availability – The availability of the primary service provided by a computing system as measured by the users of that service. This differs from general high availability which can be discussed on a component-by-component level.

Service group – A grouping of one or more components, along with their redundant counterparts, which provide a service. A set of three power supplies might form a power supply service group which supplies power to the system.

SNMP – Simple Network Message Protocol. SNMP is a set of protocols that pass information about whether or not a component is operating properly. SNMP uses ***management information bases*** (***MIB***s) to store data about a system.

Software Rejuvenation – Restarting a software component from no later than the point at which it gets allocated resources and initializes its variables.

Spatial redundancy – Redundancy by use of extra components in either hardware or software. See also temporal redundancy

Spare – The role of a redundant component that is not in standby, but is available to be placed into service by the configuration management service.

Standby – The role of a redundant component that is monitoring its redundant counterpart and is ready to be switched into service in place of its counterpart.

State – Information used by an application to determine its function and significant points of operation. State may be as simple as available vs. failed, or as complex as the frame ID for the next package to be processed along with the parameters needed for processing.

State change – Any change in the state information of an application. This term is sometimes used in the strict sense to refer only to changes in the operational state from available to in-service to faulted or similar states.

Switchover – Changing from using one component to using its redundant counterpart.

System Log – A file that keep track of major events which occur in the system. This typically includes starts, stops, faults and other similar information.

System MIB – A Management Information Base (MIB) that contains the information about basic system parameters and kernel operation.

System model – A computer-usable representation of the capabilities, characteristics and dependencies of all of the components that could be included in a system.

Temporal redundancy – Redundancy provided by re-performing operations. Most network protocols provide this form of redundancy, as raw network traffic is inherently subject to errors.

Threshold crossing – This action occurs when the value of a particular parameter being monitored crosses over a pre-determined point while either rising of falling. It is typically used to trigger a fault detector, for example a high temperature in the chassis may be used to trigger a fault detector.

TMR – see Triple modular redundant

Triggered – Setting a fault detector from showing no fault to showing a fault. Triggering a fault detector starts the process of fault management.

Triple modular redundant (TMR) – A redundant system in which there are three modules operating in parallel. If they do not all give the same information an error is detected and the output information is set to equal whatever two of the components agree upon.

Unassigned – The role of a redundant component that is not in standby, but is available to be placed into service by the configuration management service. An unassigned component is the same as a spare.

Value Coasting – Using a previous output value instead of a known bad value to avoid system failure. This is a form of fault isolation, but it must be used with notification of the failure.

Verification – The process of checking a system to ensure that design or implementation issues have not created latent faults.

Walk-over – An occurrence of one software component, or section of that component, over-writing the memory allocated to another component. One typical example of this is stack overflow, where the stack pointer start writing stack information on top of program code.

Warm Restart – Restarting a system without turning off the hardware.

WBEM – see Web-Based Enterprise Management.

Web-Based Enterprise Management (WBEM) – A standard being driven by the DMTF (Distributed Management Task Force) for managing groups of computers connected in a network.

13.0 Bibliography

Section 3:

[Ande81]
Anderson, T. and Lee, P.A., *Fault Tolerance – Principles and Practice*, Prentice-Hall, 1981.

[DHB3'00]
D. H. Brown Associates, Inc. (DHBA), *Competitive Analysis of UNIX Cluster Functionality – Part One of Two Part HA Study*, March 2000.

[Gray92]
Gray, Jim and Reuter, Andreas – *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann Publishing, San Mateo, CA, 1992.

[Lapr92]
Laprie, J.C., *Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerant Systems*, vol. 5. Springer-Verlag, Wien, New York, 1992.

[Lyu96]
Michael R. Lyu, *Software Reliability Engineering,* McGraw Hill, New York, 1996.

[Lyu96a]
Michael R. Lyu, *Software Reliability Engineering,* Chapter 6, McGraw Hill, New York, 1996.

[Rand95]
Randell, Laprie, Kopetz, and Littlewood, *Predictably Dependable Computing Systems*, Springer-Verlag Berlin Heidelberg, New York, 1995

[Wu99]
Wu, Jie, *Distributed System Design*, CRC Press, Boca Raton, Florida, 1999.

Section 5:

[X.731]
International Telecommunication Union (ITU) recommendation X.731 for State Management, 01/92

Standards:

CIM – Common Information Model
— http://www.dmtf.org/

CMIP – Common Management Information Protocol
— http://www.ietf.org/rfc/rfc1189.txt?number=1189

CMIS (common management information services).
— http://www.ietf.org/rfc/rfc1189.txt?number=1189

CORBA – Common Object Request Broker Architecture
— http://www.omg.org/gettingstarted/corbafaq.htm

IPMI – Intelligent Platform Management Interface
— http://developer.intel.com/design/servers/ipmi/

PICMG – PCI Industrial Computer Manufacturers Group
— http://www.picmg.org/

RMON – Remote Monitoring
— http://www2.ietf.org/rfc/rfc2819.txt?number=2819

SNMP – Simple Network Management protocol
— http://www.ietf.org/ids.by.wg/snmpv3.html

TMN – Telecommunications Management Network
— http://www.itu.int/TMN/

X.731 – ITU recommendation X.731 for State Management
— http://www.itu.int/itudoc/itu-t/rec/x/x500up/x731.html