SUN BLUEPRINTS
[illegible]
APPROVED

Cluster and Complex System Design Issues

By Richard Elling - Enterprise Engineering and
Tim Read - High End Systems Group

Sun BluePrints™ OnLine - November 2001

http://www.sun.com/blueprints

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303 USA
650 960-1300 fax 650 969-9131

Part No.: 816-3377-10
Revision 1.1, 11/28/01
Edition: November 2001

Cluster and Complex System Design Issues

Editor's Note – The following is the entire first chapter of the Sun BluePrints™ book, "Designing Enterprise Solutions with Sun™ Cluster 3.0." (ISBN #0-13-008458-1) which is scheduled for publication by Prentice-Hall in December 2001 and is expected to be available through www.sun.com/books, amazon.com, fatbrain.com and Barnes & Noble bookstores. Only the reference page numbers were removed from the original text. The other chapters referenced herein are only available in the printed book. This excerpt provides the reader useful information on its own merits, plus an appreciation for the contents and motivation of the remainder of the book.

This chapter addresses the following topics:

- The need for a business to have a highly available, clustered system
- System failures that influence business decisions
- Factors to consider when designing a clustered system
- Failure modes specific to clusters, synchronization, and arbitration

To understand why you are designing a clustered system, you must first understand the business need for such a system. Your understanding of the complex system failures that can occur in such systems will influence the decision to use a clustered system and will also help you design a system to handle such failures. You must also consider issues such as data synchronization, arbitration, caching, timing, and clustered system failures—split brain, multiple instances, and amnesia—as you design your clustered system.

Once you are familiar with all the building blocks, issues and features that enable you to design an entire clustered system, you can analyze the solutions that the Sun™ Cluster 3.0 software offers to see how it meets your enterprise business need and backup, restore, and recovery requirements.

The sections in this chapter are:

- Business Reasons for Clustered Systems
- Failures in Complex Systems
- Data Synchronization
- Arbitration Schemes
- Data Caches
- Timeouts
- Failures in Clustered Systems
- Summary

Business Reasons for Clustered Systems

Businesses build clusters of computers to improve performance or availability. Some products and technologies can improve both. However, much clustering activity driving the computer industry today is focused on improving service availability.

Downtime is a critical problem for an increasing number of computer users. Computers have not become less reliable, but users now insist on greater degrees of availability. As more businesses depend on computing as the backbone of their operation, around-the-clock availability of services becomes more critical.

Downtime can translate into lost money for businesses, potentially large amounts of money. Large enterprise customers are not the only ones to feel this pinch. The demands for mission-critical computing have reached the workgroup, and even the desktop. No one today can afford downtime. Even the downtime required to perform maintenance on systems is under pressure. Computer users want the systems to remain operational while the system administrators perform system maintenance tasks.

Businesses implement clusters for availability when the potential cost of downtime is greater than the incremental cost of the cluster. The potential cost of downtime can be difficult to predict accurately. To help predict this cost, you can use risk assessment.

Risk Assessment

Risk assessment is the process of determining what results when an event occurs. For many businesses, the business processes themselves are as complex as the computer systems they rely on. This significantly complicates the systems architect’s risk assessment. It may be easier to make some sort of generic risk assessment in which the business risk can be indicted as cost. Nevertheless, justifying the costs of a clustered system is often difficult unless one can show that the costs of implementing and supporting a cluster can reduce the costs of downtime. Since the former can be measured in real dollars and the latter is based on a multivariate situation with many probability functions, many people find it easier to relate to some percentage of “uptime.”

Clusters attempt to decrease the probability that a fault will cause a service outage, but they cannot prevent it. They do, however, limit the maximum service outage time by providing a host on which to recover from the fault. Computations justifying the costs of a cluster must not assume zero possibility of a system outage. Prospect theory is useful to communicate this to end users in such a situation. To say the system has “a 99 percent chance of no loss” is preferable to “a 1 percent chance of loss.” However, for design purposes, the systems architect must consider carefully the case where there is 1 percent chance of loss. You must always consider the 1 percent chance of loss in your design analysis. After you access the risks of downtime, you can do a more realistic cost estimate.

Cost Estimation

Ultimately, everything done by businesses can be attributed to cost. Given infinite funds and time (“time is money)” perfect systems can be built and operated. Unfortunately, most real systems have both funding and time constraints.

Nonrecurring expenses include hardware and software acquisition costs, operator training, software development, and so forth. Normally, these costs are not expected to recur. The nonrecurring hardware costs of purchasing a cluster are obviously greater than an equivalent, single system. Software costs vary somewhat. There is the cost of the cluster software and any agents required. An additional cost may be incurred as a result of the software licensing agreement for any other software. In some cases, a software vendor may require the purchase of a software license for each node in the cluster. Other software vendors may have more flexible licensing, such as per-user licenses.

Recurring costs include ongoing maintenance contracts, consumable goods, power, network connection fees, environmental conditioning, support personnel, and floor space costs.

Almost all system designs must be justified in economic terms. Simply put, is the profit generated by the system greater than its cost? For systems that do not consider downtime, economic justification tends to be a fairly straightforward calculation.

$$P_{lifetime} = R_{lifetime} - C_{downtime} - C_{nonrecuring} - \Sigma C_{recurring}$$

where:

$P_{lifetime}$ is the profit over the lifetime of the system.

$R_{lifetime}$ is the revenue generated by the system over its lifetime.

$C_{downtime}$ is the cost of any downtime.

$C_{nonrecurring}$ is the cost of nonrecurring expenses.

$C_{recurring}$ is the cost of any recurring expenses.

During system design these costs tend to be difficult to predict accurately. However, they tend to be readily measurable on well-designed systems.

The cost of downtime is often described in terms of the profit of uptime**.**

$$Cdowntime(t) = tdown \times \frac{Puptime}{tup}$$

where:

$C_{downtime}$ is the cost of downtime.

t_{down} is the duration of the outage.

P_{uptime} is the profit made during $t_{up.}$

t_{up} is the time the system had been up.

For most purposes, this equation suffices. What is not accounted for in this equation is the opportunity cost. If a web site has competitors and is down, a customer is likely to go to one of the competing web sites. This defection represents an opportunity loss that is difficult to quantify.

The pitfall in using such an equation is that the P_{uptime} is likely to be a function of time. For example, a factory that operates using one shift makes a profit only during the shift hours. During the hours that the factory is not operating, the P_{uptime} is zero, and consequently the $C_{downtime}$ is zero.

$$Cdowntime(t) = tdown \times \frac{Puptime(t)}{tup}$$

where:

$P_{uptime(t)} = P_{nominal}$, when t is during the work hours

= 0, all other times

Another way to show the real cost of system downtime is to weight the cost according to the impact on the business. For example, a system that supports a call center might choose impacted user minutes (IUM), instead of a dollar value, to represent the cost of downtime. If 1000 users are affected by an outage for 5 minutes, the IUM value is 1,000 users times 5 minutes, or 5,000 IUMs. This approach has the advantage of being an easily measured metric. The number of logged-in users and the duration of any outage are readily measurable quantities. A service level agreement (SLA) that specifies the service level as IUMs can be negotiated. IUMs can then be translated into a dollar value by the accountants.

Another advantage of using IUMs is that the service provided to the users is measured, rather than the availability of the system components. SLAs can also be negotiated on the basis of service availability, but it becomes difficult to account for the transfer of the service to a secondary site. IUMs can be readily transferred to secondary sites because the measurement is not based in any way on the system components.

Failures in Complex Systems

This section discusses failure modes and effects in complex systems. From this discussion you can gain an appreciation of how complex systems can fail in complex ways. Before you can design a system to recover from failures, you must understand how systems fail.

Failures are the primary focus of the systems architect designing highly available (HA) systems. Understanding the probability, causes, effects, detection, and recovery of failures is critical to building successful HA systems. The professional HA expert has many years of study and experience with a large variety of esoteric systems and tools that are used to design HA systems. The average systems architect is not likely to have such tools or experience but will be required to design such systems. Fortunately, much of the detailed engineering work is already done by vendors, such as Sun Microsystems, who offer integrated HA systems.

A typical systems design project is initially concerned with defining “what the system is supposed to do.” The systems architect designing highly available clusters must also be able to concentrate on “what the system is not supposed to do.” This is known as testing for unwanted modes, which can occur as a result of integrating components that individually perform properly but may not perform together as expected. The latter can be much more difficult and time consuming than the former, especially during functional testing. Typical functional tests attempt to show that a system does what it is supposed to do. However, it is just as important, and more difficult, to attempt to show that a system does not do what it is not supposed to do.

A *defect* is anything that, when exercised, prevents something from functioning in the manner in which it was intended. The defect can, for example, be due to design, manufacture, or misuse and can take the form of a badly designed, incorrectly manufactured, or damaged hardware or software component. An *error* usually results from the *defect* being exercised, and if not corrected, may result in a *failure.*

Examples of defects include:

- Hardware factory defect—A pin in a connector is not soldered to a wire correctly, resulting in data loss when exercised.
- Hardware field defect—A damaged pin no longer provides a connection, resulting in data loss when exercised.
- Software field defect—An inadvertently corrupted executable file can cause an application to crash.

An *error* occurs when a component exhibits unintended behavior and can be a consequence of:

- A defect being exercised
- A component being used outside of its intended operational parameters
- Some other cause, for example a random, though anticipated, environmental effect

A *fault* is usually a defect, but possibly an imprecise error, and should be qualified. "Fault" may be synonymous with bug in the context of software faults [Lyu95], but need not be, as in the case of a page fault.

Highly available computer systems are not systems that never fail. They experience, more or less, the same failure rates on a per component basis as any other systems. The difference between these types of systems is how they respond to failures. You can divide the basic process of responding to failures into five phases.

FIGURE 1 shows the five phases of failure response:

1. Fault detection
2. Fault isolation to determine the source of the fault and the component or field replaceable unit (FRU) that must be repaired
3. Fault correction, if possible, in the case of automatically recoverable components, such as error checking and correction (ECC) memory
4. Failure containment so that the fault does not propagate to other components
5. System reconfiguration so you can repair the faulty component

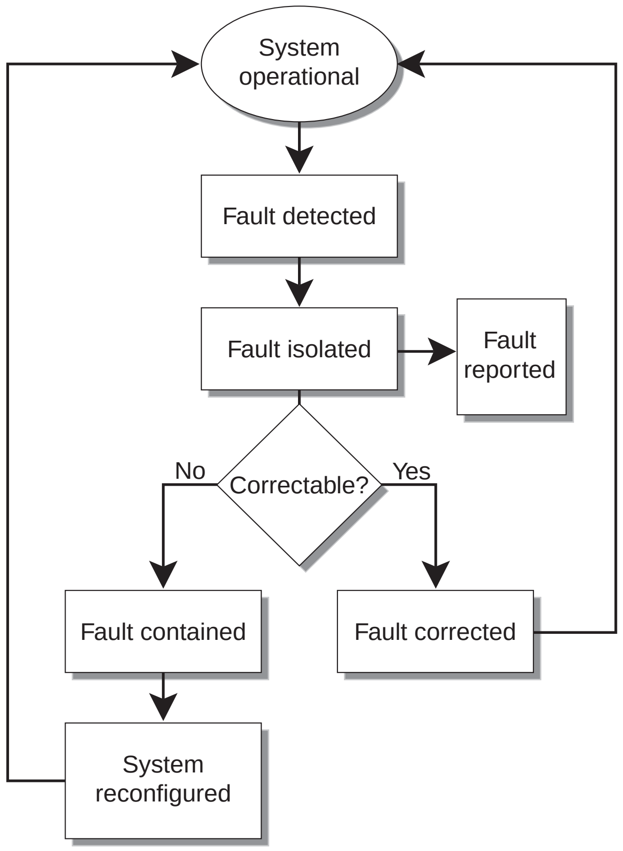

FIGURE 1 HA System Failure Response

Fault Detection

Fault detection is an important part of highly available systems. Although it may seem simple and straightforward, it is perhaps the most complex part of a cluster. The problem of fault detection in a cluster is an open problem—one for which not all solutions are known. The Sun Cluster strategy for solving this problem is to rely on industry-standard interfaces between cluster components. These interfaces have built-in fault detection and error reporting. However, it is unlikely that all failure modes of all components and their interactions are known.

While this may sound serious, it is not so bad when understood in the context of the cluster. For example, consider unshielded twisted pair, 10BASE-T Ethernet interfaces. Two classes of failures can affect the interface—physical and logical. These errors can be further classified or assigned according to the four layers of the

TCP/IP stack, but for the purpose of this discussion, the classification of physical and logical is sufficient. Physical failures are a bounded set. They are often detected by the network interface card (NIC). However, not all physical failures can be detected by a single NIC, nor can all physical failures be simulated by simply removing a cable.

Knowing how the system software detects and handles error conditions as they occur is important to a systems architect. If the network fails in some way, the software should be able to isolate the failure to a specific component.

For example, TABLE 1-1 lists some common 10BASE-T Ethernet failure modes. As you can see from this example, there are many potential failure modes. Some of these failure modes are not easily detected.

TABLE 1 Common 10BASE-T Ethernet Failure Modes

Description	Type	Detected by	Detectability
Cable unplugged	Physical	NIC	Yes, unless link test is disabled.
Cable shorted	Physical	NIC	Yes
Cable wired in reverse polarity	Physical	NIC	Yes
Cable too long	Physical	NIC (in some cases only)	Difficult because the error may range from no link (with Software Query Enable (SQE) disabled) to high bit error rate (BER) that must be detected by logical tests.
Cable receive pair wiring failure	Physical	NIC	Yes, unless SQE is enabled.
Cable transmit pair wiring failure	Physical	Remote device	Yes, unless SQE is enabled.
Electromagnetic interference (EMI)	Physical	NIC (in some cases only)	Difficult, because the errors may be intermittent with the only detection being changes in the BER.
Duplicate medium access control address (MAC)	Logical	Solaris™ Operating Environment	Yes

TABLE 1 Common 10BASE-T Ethernet Failure Modes *(Continued)*

Description	Type	Detected by	Detectability
Duplicate IP address	Logical	Solaris Operating Environment	Yes
Incorrect IP network address	Logical		Not automatically detectable for the general case.
No response from remote host	Logical	Sun Cluster software	Sun Cluster software uses a series of progressive tests to try to establish connection to the remote host.

Note – You may be tempted to simulate physical or even logical network errors by disconnecting cables, but this does not simulate all possible failure modes of the physical network interface. Full physical fault simulation for networks can be a complicated endeavor.

Probes

Probes are tools or software that you can use to detect most system faults and to detect latent faults. You can also use probes to gather information and improve the fault detection. Hardware designs use probes for measuring environmental conditions such as temperature and power. Software probes query service response or act like end users completing transactions.

You must put probes at the end points to effectively measure end-to-end service level. For example, if a user community at a remote site requires access to a service, a probe system must installed at the remote site to measure the service and its connection to the end users. It is not uncommon for a large number of probes to exist in an enterprise that provides mission-critical services to a geographically distributed user base. Collecting the probe status at the operations control center that supports the system is desirable. However, if the communications link between the probe and operations control center is down, the probe must be able to collect and store status information for later retrieval. For more information on probes, see “Failure Detection.”

Complex services require complex probes to inquire about all capabilities of the service. This complexity produces opportunity for defects in the probe itself. You cannot rely on a faulty probe to deliver an accurate status of the service.

Latent Faults

To detect latent faults in hardware, you can use special test software such as the Sun™ Management Center Hardware Diagnostic Suite (HWDS) software. The HWDS allows you to perform tests that exercise the hardware components of a system at scheduled intervals. Because these tests consume some system resources, they are done infrequently. Detected errors are treated as normal errors and reported to the system by the normal error reporting mechanisms.

The most obvious types of latent faults are those that exist but are not detected during the testing process. These include software bugs that are not simulated by software testing, and hardware tests that do not provide adequate coverage of possible faults.

Fault Isolation

Fault isolation is the process of determining, from the available data, which component caused a failure. Once the faulty component is identified or isolated, it can be reset or replaced with a functioning component.

The term fault isolation is sometimes used as a synonym for fault containment, which is the process of preventing the spread of a failure from one component to others.

For analysis of potential modes of failure, it is common to divide a system into a set of disjointed *fault isolation zones*. Each error or failure must be attributed to one of these zones. For example, a field replaceable unit (FRU) or an application process can represent a fault isolation zone.

When recovering from a failure, the system can reset or replace numerous components with a single action. For example, a Sun Quad FastEthernet™ card has four network interfaces located on one physical card. Because recovery work is performed on all components in a *fault recovery zone*, replacing the Sun Quad FastEthernet card affects all four network interfaces on the card.

Fault Reporting

Fault reporting notifies components and humans that a fault has occurred. Good fault reporting with clear, unambiguous, and concise information goes a long way toward improving system serviceability.

Berkeley Standard Distribution (BSD) introduced the `syslogd` daemon as a general-purpose message logging service. This daemon is very flexible and network aware, making it a popular interface for logging messages. Typically, the default `syslogd` configuration is not sufficient for complex systems or reporting structures. However, correctly configured, `syslogd` can efficiently distribute messages from a number of systems to centralized monitoring systems. The `logger` command provides a user level interface for generating `syslogd` messages and is very useful for systems administration shell scripts.

Not all fault reports should be presented to system operators with the same priority. To do so would make appropriate prioritized responses difficult, particularly if the operator was inexperienced. For example, media defects in magnetic tape are common and expected. A tape drive reports all media defects it encounters, but may only send a message to the operator when a tape has exceeded a threshold of errors that says that the tape must be replaced. The tape drive continues to accumulate the faults to compare with the threshold, but not every fault generates a message for the operator.

Faults can be classified in terms of correctability and detectability. Correctable faults are faults that can be corrected internally by a component and that are transparent to other components (faults inside the black box.) Recoverable faults, a superset of correctable faults, include faults that can be recovered through some other method such as retrying transactions, rerouting through an alternate path, or using an alternate primary. Regardless of the recovery method, correctable faults are faults that do not result in unavailability or loss of data. Uncorrectable errors do result in unavailability or data loss. Unavailability is usually measured over a discrete time period and can vary widely depending on the service level agreements with the end users.

Reported correctable (RC) errors are of little consequence to the operator. Ideally, all RC errors should have soft-error rate discrimination algorithms applied to determine whether the rate is excessive. An excessive rate may require the system to be serviced.

Error correction is the action taken by a component to correct an error condition without exposing other components to the error. Error correction is often done at the hardware level by ECC memory or data path correction, tape write operation retries, magnetic disk defect management, and so forth. The difference between a correctable error and a fault that requires reconfiguration is that other components in the system are shielded from the error and are not involved in its correction.

Reported uncorrectable (RU) errors notify the operators that something is wrong and give the service organization some idea of what to fix.

Silent correctable (SC) errors cannot have rate-discrimination algorithms applied because the system receives no report of the event. If the rate of an SC error is excessive because something has broken, no one ever finds out.

Silent uncorrectable (SU) errors are neither reported nor recoverable. Such errors are typically detected some time after they occur. For example, a bank customer discovers a mistake while verifying a checking account balance at the end of the month. The error occurred some time before its eventual discovery. Fortunately, most banks have extensive auditing capabilities and processes to ultimately account for such errors.

TABLE 1-2 shows some examples of reported and correctable errors.

TABLE 2 Reported and Correctable Errors

	Correctable	Uncorrectable
Reported	RC DRAM ECC error, dropped TCP/IP packet	RU Kernel panic, serial port parity error
Silent	SC Processor branch prediction table error	SU Undetected data corruption

For additional details on detectable faults, see “Fault Detection” on page 7 and “Failure Detection.”

Fault Containment

Fault containment is the ability to contain the effects and prevent the propagation of an error or failure, usually due to some boundary. For clusters, computing nodes are often the fault containment boundary. The assumption is that the node halts, as a direct consequence of the failure or by a *failfast* or *failstop*, before it has a chance to do significant I/O and propagate the fault.

Fault containment can also be undertaken proactively, for example, through failure fencing. The concept of failure fencing is closely related to failstop. One way to ensure that a faulty component cannot propagate errors is to prevent it from accessing other components or data. “Disk Fencing” describes the details of how the Sun Cluster software products use this failure fencing technique.

Fault propagation occurs when a fault in one component causes a fault in another component. This propagation can occur when two components share a common component; it is a common mode fault. For example, a SCSI-2 bus is often used for low-cost, shared storage in clusters. The SCSI-2 bus represents a shared component that can propagate faults. If a disk or host failure hangs the SCSI-2 bus (interferes with the bus arbitration), the fault is propagated to all targets and hosts on the bus. Similarly, before the development of unshielded twisted pair (UTP) Ethernet (10BASE-T, 100BASE-T), many implementations of Ethernet networks used coaxial cable (10BASE-2). The network is subject to node faults, which interfere with the arbitration or transmission of data on the network, and can propagate to all nodes on the network through the shared coaxial cable.

Another form of fault propagation occurs when incorrect data is stored and replicated. For example, mirrored disks are synchronized closely. Bad data written to one disk is likely to be propagated to the mirror. Sources of bad data may include operator error, undetected read faults in a read-modify-write operation, and undetected synchronization faults.

Operator error can be difficult to predict and prevent. You can prevent operator errors from propagating throughout the system by implementing a time delay between the application of changes on the primary and remote site. If this time delay is large enough to ensure detection of operator error, changes on the remote site can be prevented so that the fault is contained at the primary site. This containment prevents the fault from propagating to the remote site. For details, see "High Availability Versus Disaster Recovery" on page 53.

Reconfiguration Around Faults

Reconfiguring the system around faults is a technique commonly employed in clusters. A faulty cluster node causes reconfiguration of the cluster to remove the faulty node from the cluster. Any cluster-aware services that were resident on the faulty node are started on one or more surviving nodes in the cluster. Reconfiguration around faults can be a complicated process. The paragraphs that follow examine this process in more detail.

A number of features in the Solari OE allow system reconfiguration around faults without requiring clustering software. These features are:

- Dynamic reconfiguration (DR)
- Internet protocol multipathing (IPMP)
- I/O multipathing (Sun StorEdge™ Traffic Manager)
- Alternate pathing (AP)

DR attaches and detaches system components to an active Solaris OE system without causing an outage. Thus, DR is often used for servicing components. Note that DR does not include the fault detection, isolation, containment, or reconfiguration capabilities available in Sun Cluster software.

PMP automatically reconfigures around failed network connections.

I/O multipathing balances loads across host bus adapters (HBAs). This feature, which is also known as MPxIO, was implemented in the Solaris 8 Operating Environment (Solaris OE) with kernel patch 108528-07 for SPARC®-based systems and 108529-07 for Intel-based systems.

AP reconfigures around failed network and storage paths. AP is somewhat limited in capability, and its use is discouraged in favor of IPMP and the Sun StorEdge Traffic Manager.

Future plans include tighter alignment and integration between these Solaris OE features and Sun Cluster software.

Fault Prediction

Fault prediction is the process of observing a component over time to predict when a fault is likely. Fault prediction works best when the component includes a consumable subcomponent or a subcomponent that has known decay properties. For example, an automobile computer knows the amount of fuel in the fuel tank and the instantaneous consumption rate. The computer can predict when the fuel tank will be empty—a state that would cause a fault condition. This information is displayed to the driver, who can take corrective action.

Practical fault prediction in computer systems today focuses primarily on storage media. Magnetic media, in particular, has behavior that can be used to predict when the ability to store and retrieve data will fall out of tolerance and result in a read failure in the future. For disks, this information is reported to the Solaris OE as soft errors. These errors, along with predictive failure information, can be examined using the `iostat`(1M) or `kstat`(1M) command.

Unfortunately, a large number of unpredictable faults can occur in computer systems. Software bugs make software prone to unpredictable faults.

Data Synchronization

More than one copy of data is a data synchronization problem. This section describes the data synchronization issues.

Throughout this book, the concept of *ownership of data* is important. Ownership is a way to describe the authoritative owner of the single view of the data. Using a single, authoritative owner of data is useful for understanding the intricacies of modern clustered systems. In the event of failures, the ownership can migrate to another entity. “Synchronization” on page 99 describes how the Sun Cluster 3.0 architecture handles the complex synchronization problems and issues that the following sections describe.

Data Uniqueness

Data uniqueness poses a problem for computer system architectures or clusters that use duplication of data to enhance availability. The representation of the data to people requires uniqueness. Yet there are multiple copies of the data that are identical and represent a single view of the data, which must remain synchronized.

Complexity and Reliability

Since the first vacuum tube computers were built, the reliability of computing machinery has improved significantly. The increase in reliability resulted from technology improvements in the design and manufacturing of the devices themselves. But increases in individual component reliability also increase complexity. In general, the more complex the system is, the less reliable it is. Increasing complexity to satisfy the desire for new features causes a dilemma because it works against the desire for reliability (perfection of existing systems).

As you increase the number of components in the system, the reliability of the system tends to decrease. Another way to look at the problem of clusters is to realize that a fully redundant cluster has more than twice as many components as a single system. Thus, the cost of a clustered system is more than almost twice the cost of a single system. However, the reliability of a cluster system is less than half the reliability of a single system. Though this may seem discouraging, it is important to understand that the reliability of a system is not the same as the availability of the service provided by the system. The difference between reliability and availability is that the former only deals with one event, a failure, whereas the latter also takes recovery into account. The key is to build a system in which components fail at normal rates, but which recovers from these failures quickly.

An important technique for recovering from failures in data storage is data duplication.Data duplication occurs often in modern computer systems. The most obvious examples are backups, disk mirroring, and hierarchical storage management solutions. In general, data is duplicated to increase its availability. At the same time, duplication uses more components, thus reducing the overall system reliability. Also, duplication introduces synchronization fault opportunities. Fortunately, for most cases, the management of duplicate copies of data can be reliably implemented as processes. For example, the storage and management of backup tapes is well understood in modern data centers.

A special case of the use of duplicate data occurs in disk mirrors. Most disk mirroring software or hardware implements a policy in which writes are committed to both sides of the mirror before returning an acknowledgement of the write operation. Read operations only occur from one side of the mirror. This increases the efficiency of the system because twice as many read operations can occur for a given data set size. This duplication also introduces a synchronization failure mode, in which one side of the mirror might not actually contain the same data as the other side. This is not a problem for write operations because the data will be overwritten, but it is a serious problem for read operations.

Depending on the read policy, the side of the mirror that satisfies a given read operation may not be predictable. Two solutions are possible—periodically check the synchronization and always check the synchronization. Using the former solution maintains the performance improvements of read operations while periodic synchronization occurs in the background, preferably during times of low utilization. The latter solution does not offer any performance benefit but ensures that all read operations are satisfied by synchronized data. This solution is more common in fault tolerant systems.

RAID 5 protection of data also represents a special case of duplication in which the copy is virtual. There is no direct, bit-for-bit copy of the original data. However, there is enough information to re-create the original data. This information is spread across the other data disks and a parity disk. The original data can be re-created by a mathematical manipulation of the other data and parity.

Synchronization Techniques

Modern computer systems use synchronization extensively. Fortunately, only a few synchronization techniques are used commonly. Thus, the topic is researched and written about extensively, and once you understand the techniques, you begin to understand how they function when components fail.

Microprocessor Cache Coherency

Microprocessors designed for multiprocessor computers must maintain a consistent view of the memory among themselves. Because these microprocessors often have caches, the synchronization is done through a *cache-coherency* protocol. The term coherence describes the values returned by a read operation to the same memory location. Consistency describes the congruity of a read operation returning a written value. Coherency and consistency are complementary—coherence defines the behavior of reads and writes to the same memory location and consistency defines the behavior of reads and writes with respect to accesses to other memory locations. In terms of failures, loss of either coherency or consistency is a major problem that can corrupt data and increase recovery time.

UltraSPARC™ processors use two primary types of cache-coherency protocols—*snooping* and *distributed directory-based coherency.*

- *Snooping protocol* is used by all multiprocessor SPARC implementations. No centralized state is kept. Every processor cache maintains metadata tags that describe the shared status of each cache line along with the data in the cache line.

All of the caches share one or more common address buses. Each cache *snoops* the address bus to see which processors might need a copy of the data owned by the cache.

- *Distributed directory-based coherency protocol* is used in the UltraSPARC III processor. The status of each cache line is kept in a directory that has a known location. This technique releases the restriction of the snooping protocol that requires all caches to see all address bus transactions. The distributed directory protocol scales to larger numbers of processors than the snooping protocol and allows large, multiprocessor UltraSPARC III systems to be built. The ORACLE 9*i* RAC database implements a distributed directory protocol for its cache synchronization. "Synchronization" on page 99 describes this protocol in more detail.

As demonstrated in the Sun Fire™ server, both protocols can be used concurrently. The Sun Fire server uses snooping protocol when there are four processors on board and uses directory-based coherency protocol between boards. Regardless of the cache coherency protocol, UltraSPARC processors have an atomic test-and-set operation, `ldstub`, which is used by the kernel. Atomic operations must be guaranteed to complete successfully or not at all. The test-and-set operation implements simple locks, including spin locks.

Kernel-Level Synchronization

The Solaris OE kernel is re-entrant [Vahalia96], which means that many threads can execute kernel code at the same time. The kernel uses a number of lock primitives that are built on the test-and-set operation [MM00]:

- *Mutual exclusion (mutex) locks* provide exclusive access semantics. Mutex locks are one of the simplest locking primitives.
- *Reader/writer locks* are used when multiple threads can read a memory location concurrently, but only one thread can write.
- *Kernel semaphores* are based on Dijkstra's [Dijkstra65] implementation in which the semaphore is a positive integer that can be incremented or decremented by an atomic operation. If after a decrement the value is zero, the thread blocks until another thread increments the semaphore. Semaphores are used sparingly in the kernel.
- *Dispatcher locks* allow synchronization that is protected from interrupts and is primarily used by the kernel dispatcher.

Higher-level synchronization facilities, such as *condition variables* (also called *queuing locks*), that are used to implement the traditional UNIX® sleep / wake-up facility are built on these primitives.

Application Level Synchronization

The Solaris OE offers several application program interfaces (APIs) that you can use to build synchronization into multithreaded and multiprocessing programs.

The *System Interface Guide* [SunSIG99] introduces the API concept and describes the process control, scheduling control, file input and output, interprocess communication (IPC), memory management, and real-time interfaces. POSIX and System V IPC APIs are describe; these include message queues, semaphores, and shared memory. The System V IPC API is popular, being widely implemented on many operating systems. However, the System V IPC semaphore facility used for synchronization has more overhead than the techniques available in multithreaded programs.

The *Multithreaded Programming Guide* [SunMPG99] describes POSIX and Solaris OE threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding analysis tools for multithreaded programs. The threads level synchronization primitives are very similar to those used by the kernel. This guide also discusses the use of shared memory for synchronizing multiple multithreaded processes.

Synchronization Consistency Failures

Condition variables offer an economical method of protecting data structures being shared by multiple threads. The data structure has an added condition variable, which is used as a lock. However, broken software may indiscriminately alter the data structure without checking the condition variables, thereby ignoring the consistency protection. This represents a software fault that may be latent and difficult to detect at runtime.

Two-Phase Commit

The two-phase commit protocol ensures an atomic write of a single datum to two or more different memories. This solves a problem similar to the consistency problem described previously, but applied slightly differently. Instead of multiple processors or threads synchronizing access to a single memory location, the two-phase commit protocol replicates a single memory location to another memory. These memories have different, independent processors operating on them. However, the copies must remain synchronized.

In phase one, the memories confirm their ability to perform the write operation. Once all of the memories have confirmed, phase two begins and the writes are committed. If a failure occurs, phase one does not complete and some type of error handling may be required. For example, the write may be discarded and an error message returned to the requestor.

The two-phase commit is one of the simplest synchronization protocols and is used widely. However, it has scalability problems. The time to complete the confirmation is based on the latency between the memories. For many systems, this is not a problem, but in a wide area network (WAN), the latency between memories may be significant. Also, as the number of memories increases, the time required to complete the confirmation tends to increase. Attempts to relax these restrictions are available in some software products, but this relaxation introduces the risk of loss of synchronization, and thus the potential for data corruption. Recovery from such a problem may be difficult and time consuming, so you must carefully consider the long term risks and impact of relaxing these restrictions. For details on how Sun Cluster 3.0 uses the two-phase commit protocol, see “Mini-Transactions” on page 60.

Systems also use the two-phase commit for three functions—disk mirroring (RAID 1), mirrored cache such as in the Sun StorEdge T3 array and Sun StorEdge Network Data Replicator software, and the Sun Cluster cluster configuration repository (CCR.)

Locks and Lock Management

Locks that are used to ensure consistency require lock management and recovery when failures occur. For node failures, the system must store the information about the locks and their current state in shared, persistent memory or communicate it through the interconnect to a shadow agent on another node.

Storing the state information in persistent memory can lead to performance and scalability issues because the latency to perform the store can affect performance negatively. These locks work best when the state of the lock does not change often. For example, locking a file tends to cause much less lock activity than locking records in the file. Similarly, locking a database table creates less lock activity than locking rows in the table. In either case, the underlying support and management of the locks does not change, but the utilization of the locks can change. High lock utilization is an indication that the service or application will have difficulty scaling.

An alternative to storing the state information in persistent memory is to use shadow agents—processes that receive updates on lock information from the owner of the locks. This state information is kept in volatile, main memory, which has much lower latency than shared, persistent storage. If the lock owner fails, the shadow agent already knows the state of the locks and can begin to take over the lock ownership very quickly.

Lock Performance

Most locking software and synchronization software provide a method for monitoring their utilization. For example, databases provide performance tables for monitoring lock utilization and contention. The `mpstat`(1m), `vmstat`(1m), and `iostat`(1m) processes give some indications of lock or synchronization activity, though this is not their specialty. The `lockstat`(1m) process provides detailed information on kernel lock activity, monitors lock contention events, gathers frequency and timing data on the events, and presents the data.

Arbitration Schemes

Arbitration is the act of deciding. Many forms of arbitration occur in computer systems. I/O devices arbitrate for access to an I/O bus. CPUs arbitrate for access to a multiprocessor interconnect. Network nodes arbitrate for the right to transmit on the network. In clusters, arbitration determines which nodes are part of the cluster. Once they are part of the cluster, arbitration determines how the nodes host the services. To accurately predict the behavior of the cluster and services in the event of a failure, you must understand the arbitration schemes used in the cluster.

Asymmetric Arbitration

Asymmetric arbitration is a technique commonly used when the priority of competing candidates can be established clearly and does not change. An example of this is the SCSI-2 protocol. In SCSI-2, the target address specifies the priority. If multiple targets attempt to gain control of the bus at the same time, the SCSI-2 protocol uses the target address to arbitrate the winner.

By default, the Solaris OE sets the host SCSI target address to 7, the highest priority. This helps ensure the stability of the bus because the host has priority over all of the I/O slave devices. Additional stability in the system is ensured by placing slow devices at a higher priority than fast devices. This is why the default CD-ROM target address is 6 and the default disk drive target addresses begin with 0.

SCSI priority arbitration creates an interesting problem for clusters using the SCSI bus for shared storage. Each address on the bus can be owned by only one target on the bus. Duplicate SCSI target addresses cause a fault condition that is difficult to detect and yields unpredictable behavior. For simple SCSI disks, each node of the cluster must have direct access to the bus. Therefore, one node must have a higher priority than the other node. In practice, this requirement rarely results in a problem, because the disks themselves are the unit of ownership and both nodes have higher priority than the disks.

Asymmetric arbitration is also used for Fibre Channel Arbitrated Loop (FC-AL). FC-AL is often used instead of the SCSI bus as a storage interconnect because of its electrical isolation (electrical faults are not propagated by fiber), its long distance characteristics (components can be separated by many kilometers), and its ability to be managed as a network. The priority of FC-AL nodes or ports is based on the physical loop address. Each candidate that wants to send data must first send an arbitration request around the loop to all other candidates. Once the port that is receiving the arbitration request approves and detects it, the candidate can send data. An obvious consequence is that greater numbers of candidates increase the arbitration time. Also, the candidates can be separated by several kilometers, resulting in additional latency, and the time required to arbitrate may significantly impact the amount of actual throughput of the channel.

Symmetric Arbitration

Symmetric arbitration is used when the candidates are peers and a priority-based arbitration is not used. This case is commonly found in symmetric multiprocessor (SMP) systems and networks. Arbitration is required so that all candidates can be assured that only one candidate has ownership of the shared component.

10BASE-T and 100BASE-T Ethernet networks use a carrier sense, multiple access, with collision detection (CSMA/CD) arbitration scheme. This scheme allows two or more nodes to share a common bus transmission medium [Madron89]. The node listens for the network to become idle, then begins transmitting. The node continues to listen while transmitting to detect collisions, which happen when two or more nodes are transmitting simultaneously. If a collision is detected, the node will stop transmitting, wait a random period of time, listen for idle, and retransmit. Stability on the network is assured by changing the random wait time to increase according to an algorithm called *truncated binary exponential backoff*. With this method it is difficult to predict when a clean transmission will be possible, which makes 10BASE-T or 100BASE-T Ethernet unsuitable for isochronous workloads on networks with many nodes. Also, many busy nodes can result in low bandwidth utilization.

The faster versions of Ethernet, 1000BASE-SX and 1000BASE-T (also known as Gigabit Ethernet), are only available as full duplex, switched technologies, thereby eliminating the CSMA/CD arbitration issues on the common medium. For larger networks, the arbitration role is moved into the network switch. Direct, node-to-node connections are full duplex, requiring no arbitration.

Voting and Quorum

Voting is perhaps the most universally accepted method of arbitration. It is time tested for many centuries in many forms—popularity contests, selecting members of government, and so forth. This is the method used by Sun Cluster software for arbitration of cluster membership (see “Majority Voting and Quorum Principles”).

One problem with voting is plurality—the leading candidate gains more votes than the other candidates, but not more than half of the total votes cast. Many different techniques can be used during these special cases—run-off elections, special vote by a normally nonvoting member, and so forth. Another problem with voting is that ties can occur.

A further problem with voting is that it can be time consuming. The act of voting as well as the process of counting votes is time consuming. This may not scale well for large populations.

Occasionally, voting is confused with a quorum. They are similar but distinct. A vote is usually a formal expression of opinion or will in response to a proposed decision. A quorum is defined as the number, usually a majority of officers or members of a body, that, when duly assembled, is legally competent to transact business [Webster87]. Both concepts are important; the only vote that should ratify a decision is the vote of a quorum of members. For clusters, the quorum defines a viable cluster. If a node or group of nodes cannot achieve a quorum, they should not start services because they risk conflicting with an established quorum.

Data Caches

Data caching is a special form of data duplication. Caches make copies of data to improve performance. This is in contrast to mirroring disks, in which the copies of data are made primarily to improve availability. Although the two have different goals, the architecture of the implementations is surprisingly similar.

Caches are a popular method of improving performance in computer systems. The Solaris OE essentially uses all available main memory, RAM, as a cache to the file system. Relational database management systems (RDBMS) manage their own cache of data. Modern microprocessors have many caches, in addition to the typical one to three levels of data cache. Hardware RAID arrays have caches that increase the performance of complex redundant arrays of independent disks (RAID) algorithms. Hard drives have caches to store the data read from the medium. The use of these forms of caching can be explained by examination of the cost and latency of the technology.

Cost and Latency Trade-Off

All memories are based on just a few physical phenomena and organizational principles [Hayes98]. The features that enable you to differentiate between memory technologies are cost and latency. Unfortunately, the desire for low cost and low latency are mutually exclusive. For example, a dynamic RAM (DRAM) storage cell has a single transistor and a capacitor. A static RAM (SRAM) storage cell has four to six transistors (a speed / power trade-off is possible here). The physical size of the design has a first order impact on the cost of building integrated circuits. Therefore, the cost of building N bits of DRAM memory is less than SRAM memory, given the same manufacturing technology. However, DRAM latency is on the order of 50 ns versus 5 ns for SRAM.

The cost / latency trade-off can be described graphically. FIGURE 2 shows the latency versus cost for a wide variety of technologies used in computer systems design. From this analysis, it is clear that use of memory technologies in computer system design is a cost versus latency trade-off. Any technology above the shaded area is not likely to survive due to costs. Any technology below the shaded area is likely to spawn radical, positive changes in computer system design.

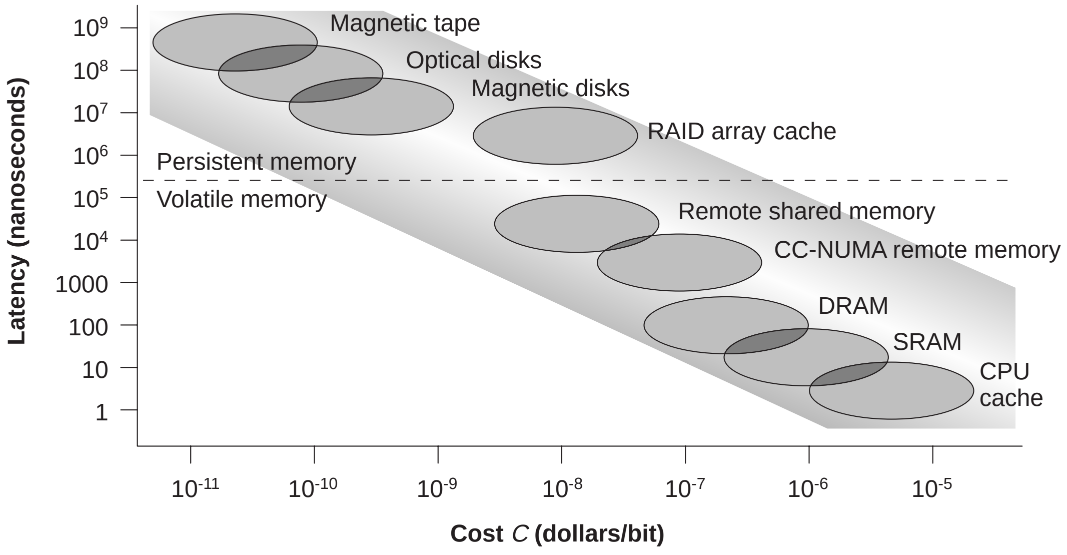

FIGURE 2 Cache Latency Versus Cost

The preceding graph also shows when caches can be beneficial. The memory technologies to the left, above the dashed line, tend to be persistent, while the memory technologies to the right, below the dashed line, tend to be volatile. Important data that needs to survive an outage must be placed in persistent memory. Caches are used to keep copies of the persistent data in low-latency memory, thus offering fast access for data that is reused. Fortunately, many applications tend to reuse data—especially if the “data” are really “instructions.” A given technology can be effectively cached by another technology having lower latency, such as those to the right.

Cache Types

CPU caches are what most computer architects think about when they use the term *cache*. While CPU caches are an active source of development in computer architectures, they are only one of the caches in a modern SMP system. CPU caches are well understood and exhibit many of the advances in computer architecture.

Metadata caches store information about information. On disk, the metadata is duplicated in multiple locations across the disk. In the Solaris OE UNIX file system (UFS), metadata is cached in main memory. The kernel periodically flushes the UFS metadata to disk. This metadata cache is stored in volatile main memory. If a crash occurs, the metadata on disk may not be current. This situation results in a file system check, `fsck`, which reconciles the metadata for the file system. The logging option introduced with UFS in the Solaris 7 OE stores changes in the metadata, the log of metadata changes, on the disk. This dramatically speeds up the file system check because the metadata can be regenerated completely from the log.

File system read caches are used in the Solaris OE to store parts (pages) of files that have been read. The default behavior for the Solaris OE is to use available RAM as a file system cache. When the Solaris 8 OE system gets low on available memory, the system discards file system cached pages in preference to executable files or text pages. Similar behavior can be activated in the Solaris 2.6 or Solaris 7 OE by enabling `priority_paging` in the `/etc/system` file.

Buffer caches provide an interface between a fast data source and a slow data sink. This type of cache is quite common, especially in I/O subsystem design. A buffer design is commonly used to implement I/O subsystems. For instance, modern SCSI or FC-AL disk drives have 8 to 16 megabytes of track cache. Track cache is really a read buffer cache designed to contain the data on at least one track of the disk media. This design allows significant improvements in performance of read operations that occur in the same track because the media requires only one physical read. The latency for physically reading data from disk media depends on the rotation speed of the media, and is often in the 5 to 10 millisecond range. Once the whole track is in the buffer, the data can be read in the latency of a DRAM access, which is on the order of 200 nanoseconds.

File systems often have a buffer cache in main memory for writing the data portion of files. These write buffers are distinguished from the file system read caches in that the write cache pages cannot be discarded when available memory is low. UFS implements a high-water mark for the size of its write buffer cache on a per file basis that can be tuned with the *ufs:ufs_HW* variable in the `/etc/system` file. The memory used for write buffer caches can cause resource contention for memory when the available RAM is low. For low memory situations, the use of write buffer cache should be examined closely.

Cache Synchronization

What all caches have in common is that the data must be synchronized between the copy in the low-latency memory and the higher-latency memory. Sequential access caches, such as the buffer cache, have relatively simple synchronization—whatever comes in, goes out in the same order. Random access caches such as CPU, metadata, and file system read caches can have very complex synchronization mechanisms.

Random access caches typically have different policies for read and write. Since the cost per bit of caches is higher than that of higher-latency memories, the caches tend to be smaller. All of the data that can be stored in the higher-latency memory cannot fit in the cache. The cache policy for reading data is to discard the data, if needed, because it can be read again in the future. The cache policy for writing cannot simply discard data. Write policies tend to belong to two categories—write-through and write-behind. A write-through policy writes the data to the lower-latency memory, immediately ensuring that the data can be safely discarded if necessary. A write-behind policy does not immediately write the data to the higher-latency memory, thus improving performance if the data is to be written again soon. Obviously, the write-behind policy is considerably more complicated. For this, the hardware or software must maintain information about the state of the data, that is, whether or not the data has been stored in the higher-latency memory. If not, then the data must be written before it can be discarded to make room for other data.

If the persistence requirement of the cache and its higher-latency memory is the same, either both are persistent or not, write-behind cache policies make good sense. If the cache is not persistent and the higher-latency memory is persistent, write-through policies provide better safety.

As can be expected, the increasing complexity in cache designs increases the number of failure modes that can affect a system. At a minimum, caches add hardware to the system, thereby reducing the overall system reliability. Alternatively, caches use existing storage resources that are shared with other tasks, potentially introducing unwanted modes. For every synchronization mechanism added, an arbitration mechanism must also be added to handle conflicts and failures. A good rule of thumb is to limit the dependence on caches when possible. This trade-off decision must often be made while taking into account the benefits of caching. You should have a firm understanding of the various caches used in a system. To help with the decision making process, you can use the example in FIGURE 2, which shows the caching hierarchy of the system.

Timeouts

Computer systems use timeouts to help maintain state information. Since digital computers are discrete systems, you must use some mechanism to create an event and to verify that an event was not completed. You can use timeouts for cluster heartbeats, arbitration cycles to determine if a processor has hung, and so forth.

Caution – Reducing the size of timeouts in an effort to improve the ability to detect component failures is tempting. You must avoid destabilizing conditions in which the timeout values are too small. Ensuring that the system remains stable when timeouts are changed requires due diligence and detailed engineering analysis.

Timeouts are also opportunities for defects. In particular, the stability of a system can be undermined if you design a series of nested timeouts incorrectly. FIGURE 3 is a nested timing diagram example.

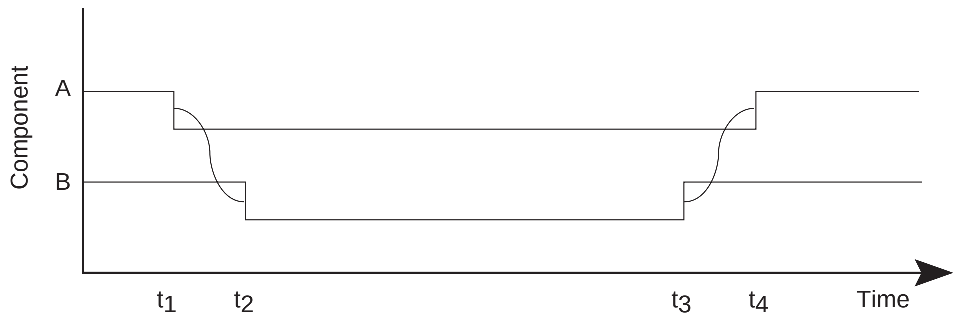

FIGURE 3 Nested Timing Diagram—Example

In this example, component A has a state change at time t_1 that causes a state change in component B at time t_2. Component B then has a state change at time t_3 that causes a state change in component A at t_4. This timing diagram example is analogous to a host, A, issuing a read data command to a disk drive, B. This system has deterministic, sequential behavior. However, if a failure occurs in component B between t_2 and t_3, the state change of component A at t_4 will never occur. Component A will hang. If a timeout is implemented, component A can detect the failure of component B. FIGURE 4 shows the failure of B and the point at which A times out.

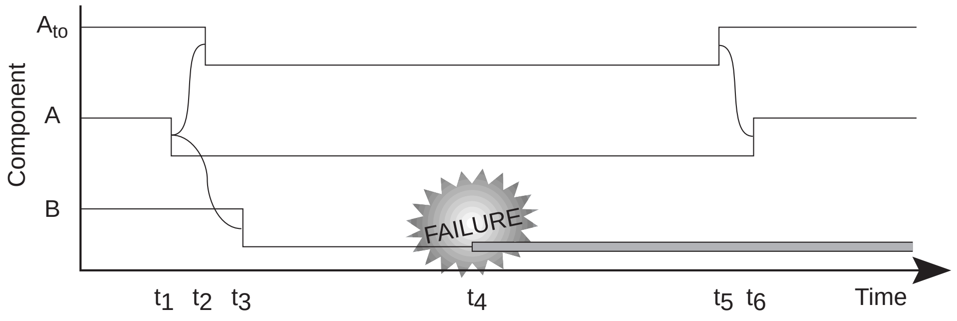

FIGURE 4 Nested Timing Diagram With Timeout

In this case, A implements an internal timeout, A_{to}. At time t_1, component A starts the timeout counter, A_{to}, at t_2 and causes a state change in B at t_3. At time t_4, component B fails, never to recover. At time t_5, the A_{to} timeout expires, causing A to change its state. Since both A and A_{to} are part of component A, A knows an error condition occurred in component B.

Stable System

FIGURE 5 shows a stable system implementing timeouts. In this case, the component timeout of A is greater than the service time of component B.

Timeout for Ato = (t6 - t2) > (t4 - t3):

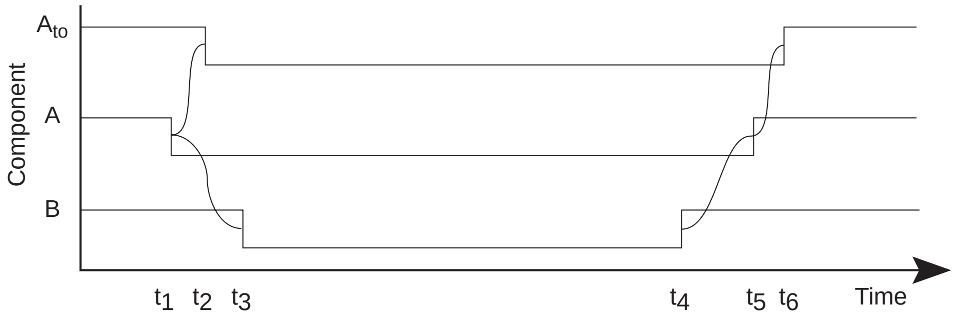

FIGURE 5 Stable System With Timeout

Unstable System

FIGURE 6 shows an unstable system implementing timeouts. In this case, the timeout value component A is less than the service time of component B.

Timeout for Ato = (t4 - t2) > (t6 - t3):

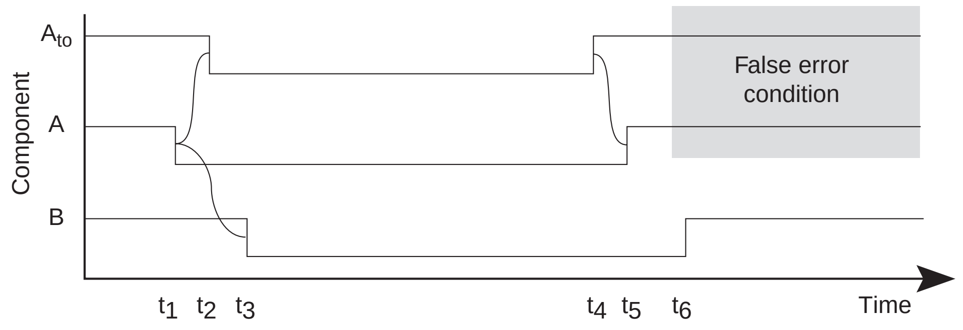

FIGURE 6 Unstable System With Timeout

The system is unstable because it detects false errors. The action taken by component A, because of the presumed failure of component B, determines the stability of the overall system.

Stability Problems

The *stability problems* inherent in complex computer systems using timeouts are difficult to predict or prove mathematically because of the multivariate and nonlinear nature of these systems.

Timeouts that are too long increase the time to detect errors. Timeouts that are too short generate false error conditions. False error conditions cause unnecessary failovers.

Timeouts are a source of systems integration errors. For components that use timeouts, the default timeout values are set according to their expected uses. When combined to build a larger system, the timeouts can cause the system instability. You must understand the component timeouts and their effect on event detection.

Failures in Clustered Systems

A number of faults are specific to clustered systems. These faults affect synchronization between nodes, arbitration of conflicts regarding shared components, or the presentation of a single view of data or services to external users and systems. “Cluster Failures” on page 91 describes how Sun Cluster 3.0 software avoids and handles these faults. The sections that follow describe the split brain, multiple instance, and amnesia faults.

Split Brain

Split-brain failure occurs when a single cluster has a failure that results in reconfiguration into multiple partitions, with each partition forming without knowledge of the existence of any other. If each new cluster does not know of the existence of the others, there could be a collision in shared resources. Network addresses can be duplicated because each of the new clusters thinks it should own the shared network address. Shared storage could also be affected, because each cluster believes that it owns the shared storage. The result is *severe data corruption*.

“Amnesia and Temporally Split Configurations” on page 90 explains how Sun Cluster 3.0 software avoids the split-brain condition.

Multiple Instances

A multiple instance failure occurs when an application is designed to operate on data assuming it has *exclusive access* to the data and more than one instance of the application is started. For a single system there are many ways to prevent this failure, such as semaphores, checking the process list, creating lock files, and so forth. In a clustered environment, this prevention becomes more difficult. Using semaphores and checking the process list only applies to the local node. Creating lock files works, but only if the lock files are on shared storage. Even then, a simple lock file will not suffice, because the application must have a mechanism to correctly query each node in the cluster to ensure that another instance of the application is actually running, since the application may have failed without releasing the lock.

Amnesia

Amnesia is a failure mode in which a node starts with stale cluster configuration information. This is a synchronization error because the cluster configuration information is not propagated to all of the nodes. For example, if a node fails and the cluster is then reconfigured, cluster configuration information of the node is now stale. If the node tries to rejoin the cluster, the node must resynchronize its cluster configuration information.

A more difficult situation occurs when the node fails, the cluster is reconfigured, the cluster is brought down, and then the failed node is brought back up. In this case, the stale cluster configuration information is presumed to be correct, and a new cluster is built with the stale configuration information.

Summary

Many businesses implement highly available clusters when the risk of the costs of downtime exceeds the costs of the cluster. This can be quantified in business terms and measured by well-designed systems.

We examined how failures occur in complex systems and showed methods that contain, isolate, report, and repair failures. Synchronization is used by any component or system that creates copies of data, including redundant storage, caches, and cluster components. Arbitration is important in clustered systems for deciding which of many components is the appropriate configuration to provide services.

Finally, we examined special considerations for clustered systems. Many systems use caches to improve performance. Caches introduce synchronization problems because they represent duplication of data. Timeouts are used to try to determine if a component has failed, but care must be taken to ensure that timeouts are properly tuned.

A number of failure modes are specific to clusters—split brain, amnesia, and multiple instances. Clusters use special techniques to help prevent such failures.

Author's Bio: Richard Elling

Richard Elling is the Chief Architect for Enterprise Engineering at Sun Microsystems in San Diego, California. Richard had been a field systems engineer at Sun for five years. He was the Sun Worldwide Field Systems Engineer of the Year in 1996. Prior to Sun, he was the Manager of Network Support for the College of Engineering at Auburn University, a design engineer for a startup microelectronics company, and worked for NASA doing electronic design and experiments integration for Space Shuttle missions.

Author's Bio: Tim Read

Tim Read is a Lead Consultant for the High End Systems Group in Sun UK's Joint Technology Organization. Since 1985 he has worked in the UK computer industry, joining Sun in 1990. He holds a BSc in Physics with Astrophysics from Birmingham University. As part of his undergraduate studies Tim studied clusters of Suns; now he teaches and writes about Sun clusters.