Failure Scenarios and Mistakes Commonly Found in Distributed Systems

By Dr. Mark Hayden, CTO, Ventura Networks, Inc.
October 23, 2003

Introduction

This document describes many of the issues and pitfalls to be considered when developing distributed systems and explains how Ventura Network's Precision Suite of Products helps developers avoid these problems. These pitfalls consist of many errors or unaddressed issues that distributed systems commonly fall victim to. After reading this document, you should have a better understanding of why distributed systems are considered to be the most complex kind of software to develop and why it is so important and beneficial to leverage commercially available technologies to facilitate their development.

When reading this document, it is important to understand that there are two primary descriptions for the fault tolerance of a distributed system: *data integrity* and *high availability*.

- Data integrity relates to whether a system protects its configuration and other data from becoming corrupted in such a way that would cause a loss of data or a disruption of service. A simple example of a data integrity failure is a data replication system in which the replicas allow inconsistent changes to be made. More complex examples include cases where the system becomes confused and begins acting erratically because of an inability to cope with behavior in the network. Distributed systems require extreme levels of data integrity, otherwise business continuation is put at serious risk and significant amounts of money can be lost.
- High availability relates to whether a system will be able to continue operating in the presence of one or more failures, either in the network or in the machine. By examining the different types of failures that can be sustained and how systems monitor and respond to them, it is possible to determine the fraction of time a system will be operational. For example, a system with five nines availability (*0.99999*) will be operational *99999/100000* of the time, thus, being able to compensate very quickly for issues as they arise in a network.

While both of these characteristics are important, it is useful to distinguish between them for two reasons. First, loss of data integrity is often more disruptive than loss of availability. Consider for example that if the configuration data for a distributed network management infrastructure becomes corrupted because of a temporary network perturbation, it can cause the outage to continue well beyond recovery of the hardware. Second, while availability can always be lost as a result of sufficient network failures, it is possible to develop distributed applications that never suffer from data integrity errors because of network behavior. What this means is that by designing your solution appropriately, a whole class of failures can be eliminated from a distributed system. Ventura Network's Precision Suite of Products facilitates this sort of design.

The two products discussed in this document are Ventura Network's FT*m* and Echo products. FT*m* is a Fault Tolerant Middleware product that provides a comprehensive set of communication and fault-tolerant services which facilitate the development of distributed systems. Echo is a full-featured data replication product that is also integrated as a component of FT*m*. These products are unique in the marketplace in that they address all of the issues described in this document.

1. Split Brain

This is a classic problem associated with distributed systems. It arises when machines providing certain kinds of fault tolerant services lose communication with each other but decide to continue operating independently. When different parts of the system diverge, the various groupings of machines may decide that the other groups have failed, even though such groups are actually still running and servicing requests. When this happens, both groups of machines may start to operate authoritatively for the service they provide causing contradictory changes to occur.

For example, consider a group of machines implementing a fault-tolerant storage service that lose the ability to communicate. If both sets continue accepting changes to the data they are managing, they have gone “split brain.” These machines may start accepting conflicting changes to the data they are managing which typically results in massive or total corruption of such data.

Ventura’s Echo data replication product addresses this issue by providing online quorum management that guarantees that at most one partition of the system will think it has a "quorum." Only that part of the system is allowed to continue making changes to the replicated data. The other parts are informed that they do not have a quorum and may not proceed until they can again communicate with the rest of the machines. Note that even in this state, such unconnected machines are allowed to read their copy of the data, with the caveat that it may be out of date. On reconnection to the rest of the system, all machines will be safely brought fully up to date.

2. Inconsistent Failure Detection

A fundamental aspect of distributed applications is that failures in different components of the system must be accurately detected so that countermeasures or reconfigurations can be made to adjust to the changes in the system. Problems arise when the various components of the system detect different sets of failures or detect the failures in different orders.

Consider the case of a system consisting of 5 machines, A, B, C, D, and E. In this example, D initially suffers a failure, then E fails permanently, and then D recovers from its failure. If A, B, and C are independently detecting failures, it is possible that they all observe a different series of events. Machine A could (accurately) see D fail, E fail, and then D recover. Machine B could see E fail first, D fail, and then D recover. Finally, C might see only E fail, but not detect that D had failed. This is a relatively simple scenario that does not take into account complex network failures and partitioning that could occur. Not handling these cases carefully can result in serious problems.

It is common for distributed systems to be designed assuming that failure detection is consistent throughout the system. When this assumption is made, the system will usually fail in strange and harmful ways. For instance, if in the example above, machines A, B, and C are supposed to compensate for the failures of the other machines, the different observations about failures could cause them to take inconsistent actions and corrupt data.

FT*m* addresses these issues by informing the applications on each currently running machine about the other machines still operating in the system. The applications are each presented with consistent information, regardless of the failures that occur. In particular, if some machines fail, the remaining machines are guaranteed to be presented with the same sequences of configurations. If the system is partitioned into multiple sets of machines, the different sets each see the membership constrict to the individual set they are in. When the network heals, they will then merge back together again. By receiving a consistent picture of the system as the failures occur, the applications running on the machines have a much easier job of managing failures.

3. Site Failures

A site failure occurs when all machines at a given site stop operating. A common cause of site failures is the loss of power to all machines. Note that even with uninterruptable power supplies (UPS), it is possible for a power loss to last longer than the UPS can provide power to the computers. It is not uncommon for distributed systems to be designed based on the assumption that site failures do not occur. This is often done because a site failure is classified as a "double-fault" because more than one machine has failed and many distributed systems are built with the ability to only handle single component failures. However, site failures do occur and distributed systems must be able to address them.

Systems that are not designed to handle site failures may be forced to resort to heuristics to determine which was the last machine to fail (assuming that such machine has the most up-to-date data). This guess-work is fraught with the potential for major errors that can cause serious data integrity issues.

It is also important to realize that site failures (especially from power loss) are often accompanied by other component failures. For instance, power loss often coincides with power surges and other anomalous behavior that tends to make other failures more likely. For instance, a surge could cause a failure of a power supply in one or more machines (even if protected by surge protectors and the like), or cause an ailing component such as a disk drive to suffer a hard failure. Thus, when the site comes back up, there may be one or more components that have suffered hardware failures of one kind or another. Although many engineers may consider these double-faults, they are really the result of a single event that causes other events to occur. In any case, system reliability requires that these kinds of failures be properly addressed.

Ventura’s Echo data replication product addresses site failures in a number of ways. First of all, because of Echo's multi-phase commit protocol, resynchronization protocol, and the fact that it carefully journals all information to disk prior to completing each operation, it operates correctly in the presence of site failures and is always able to guarantee correct behavior when the site recovers. After recovery, without a majority of the machines in operation, Echo will indicate to the application that it has data that is potentially out of date and let the application decide what to do, the safest choice being to wait for missing machines to come back up. Finally, Echo supports multi-site replication, including over MANs and WANs. This allows configurations with multiple sites to be supported as simply as ones with a single site.

4. Delayed messages

This problem arises in scenarios where the delivery of messages are not synchronized with the detection of failures that occur in the system. Because messages are sent and received in particular configurations, it can cause problems when messages sent from one configuration are received by a machine that is in another configuration. Oftentimes major components of applications are implemented under the assumption that the configuration in the system will not change. Of course, serious problems will arise when it does change and the messages happen to cross configuration boundaries.

For instance, a message may be sent from one machine, A, to another, B, to tell B to send a particular set of requests to A. After A fails, the responsibility for those requests may have been reallocated to another machine, C, who had informed B of the new configuration. If A’s original message to B were delayed until this point, B may become confused and send requests to A, which in turn may not be able to process the requests, or, if A has recovered, may actually mistakenly perform the requests for B, even though it should not.

As with many of the problems described in this document, delayed messaging scenarios can cause a distributed system to fail in a manner which is very hard to diagnose and test. In addition, because networks often do not behave in this fashion, developers typically do not have this in mind when developing applications. However, these kinds of problems do occur, especially in MAN and WAN configurations where the likelihood of variability in network latency increases dramatically.

FT*m* addresses delayed messages by coordinating message delivery with system configuration monitoring. FT*m* tells the application machines what the other machines in the system are doing, and will only deliver messages from machines in the applicable configuration. Furthermore, if a machine leaves the system and later recovers, FT*m* will automatically discard messages sent by that machine before the failure. It may initially seem counterintuitive for reliable communication middleware to deliberately *drop* messages, but the messages in question will normally just cause problems to the system.

5. Issues Arising While Machines Are Being Added and Removed

This issue applies most directly to data replication. Almost every system that uses replicated data must be able to change the sets of machines used for the replicas. For example, failed machines need to be replaced and occasionally new ones added to systems in the field. The potential for failures and other perturbations is significant during the process of changing the set of replicas because it involves extremely complex issues. If these issues are not addressed correctly, it leaves the system open to data integrity issues or other kinds of failures. Many distributed systems assume that nothing will happen if there are disturbances during this process and throw their hands up if some failure occurs, which typically can then result in a loss of data integrity.

For example, a set of 3 machines, A, B, and C, may be replicating configuration data for a service they are providing. If machine C suffers a hard failure, the remaining machines, A and B, can continue operating, but as soon as possible, the failure should be repaired by adding another machine, D, to the system in place of C. During this change, it is important to transfer the state of the system to D while A and B continue to replicate it. The replication protocol must continue to allow changes to the state of the system while the new machine, D, is added in to replace C. If there are any network failures or temporary machine failures during this process, it is necessary that the system not get confused or allow the data to be corrupted.

The difficulty of accomplishing the process of changing replicas safely often results in distributed systems being designed in a manner that requires all nodes be shut down before they are updated with the new replica set. This is often a complex and error prone process requiring manual intervention. It also reduces the availability of the application due to the down time that it requires. In addition, with many systems, especially embedded systems, it is just not practical to do.

Echo supports changing of replica sets and provides a simple API to add and remove machines from a system. Echo handles all of the complexity involved in this process so that the application developers do not have to. Echo-based applications therefore allow online automatic replica changes as a matter of course.

6. Failure Cycling

This problem occurs when a component in the system suffers repeated failures. An example of this is where a machine develops a regularly recurring catastrophic problem. In such a case, that machine or process may repeatedly fail and restart, only to fail again as it is coming back into the system. The repeated failure and resynchronization of a component causes significant disruption to the overall system. This can so disrupt the overall system that it is able to make only minimal progress, or in some cases no progress at all.

There are of course many ways for systems to develop these symptoms. Consider the case of a machine with a faulty network card that causes the machine to fail when it becomes very active. Shortly after powering up, the machine will contact the other machines in the network and begin to resynchronize itself with them. At this point, the load on the network card causes the machine to crash. And the process starts again.

The danger with these problems is not that the system will suffer a loss of data integrity. Rather it is that the cycling of one part of the system can cause the others to constantly try to resynchronize with the failed component and be unable to complete useful work. This reduces the effective availability of the system.

Support for avoiding these scenarios is available in FT*m* as it tracks components that are repeatedly cycling and isolates them from the remainder of the system.

7. Distributed Congestion

Congestion can occur in distributed systems in both the network and in the machines communicating over the network. Congestion in the network occurs when too many messages are being transmitted which ultimately overloads the fabric and causes the effective network performance to decrease. Additionally, the machines communicating in a distributed system can become overloaded when the amount of work they are processing becomes too large. The worst form of congestion occurs when the distributed communication has a feedback loop that causes the congestion to feed on itself and grow increasingly worse.

Many developers are accustomed to relying on TCP sockets to implement flow control. When an application attempts to write too much data to a socket, the socket causes the application to block until the receiver has read more data from the "other end" of the socket. This works well in many cases with point-to-point applications. However in distributed applications, it becomes increasingly possible for circular dependencies to develop where a set of processes are waiting forever in a loop on each other, a condition called deadlock.

Flow control is an issue that must be kept in mind when designing applications. Ultimately, it is up to the application to throttle back its communication when part of the system is becoming overloaded. That said, FT*m* supports the application by providing it with guidance when one or more communication channels in the system are becoming overloaded as well as when the overload has cleared up.

8. Changing Network Configuration

Networks have many kinds of configuration parameters that tend to change in a variety of ways. Applications deployed in one network configuration in many cases will find that the network configuration has changed out from underneath them. Attempting to handle these changes creates a great deal of complexity that is best addressed by software designed specifically to handle such issues.

For instance, consider systems operating in a DHCP (Dynamic Host Configuration Protocol) configured network, in which a DHCP server leases network addresses to its components. This means that a machine can have its network addresses change from one reboot to the next. A system using a fixed configuration will be vulnerable to breaking in such cases especially if there are other machines configured with its IP address.

FT*m* addresses this by providing automatic network configuration, tracking, and auto-discovery. FT*m* will automatically probe the system to determine the network configuration and configure itself appropriately. Additionally, FT*m* was designed from the ground up to separate the notions of physical and logical addresses. Applications always use logical addresses when interacting with FT*m* and FT*m* translates these into physical addresses as necessary so that the application is buffered from those configuration changes. The result of this is that network configuration changes are handled automatically by FT*m*.

9. Assuming the Network is Perfect

Certain distributed systems rely on redundancy in the network to guarantee not only high availability but also data integrity. When a distributed system is designed in such a way that it relies on the existence of a reliable underlying network for data integrity purposes, significant issues arise. This is because distributed systems are highly available or fault tolerant largely because they are able to properly handle problems when and as they arise in the network. For instance, when the components in such a system cannot communicate with each other because of a problem in the network, the application will fail and data can become corrupted.

An example of this design pitfall arises in many systems that replicate data amongst a group of nodes. Applications designed with the assumption that the network is perfect require that multiple network fabrics are used to prevent, for instance, split brain systems. Without the additional network connections, the inability of one group of nodes to communicate would potentially cause the system to go "split brain" and corrupt the data they are replicating. Thus, the multiple networks are there not just to make the loss of communication increasingly less likely, but are required in order to protect against the system "melting down" due to a single network failure.

Having multiple network connections increases the availability of the system by protecting against many kinds of failures, but there may still be single points of failure that can cause the network to fail. Designing a system based on the assumption of a perfect network introduces many places where subtle configuration errors can effectively introduce single points of failure that would cause data corruption. For instance, multiple network interface cards may be on a shared bus that itself has single points of failure that could cause all cards to lose connectivity. Or a software error in the networking code such as a buffer starvation could effectively eliminate communication. Network switches that mistakenly share the same power source introduce a single point of failure (power) that can cause data loss. Other examples include seemingly innocuous configuration changes such as moving a network connection to a different switch port which introduces a window of time where a single point of failure could cause data corruption. Certainly, great efforts can be made to address these issues, however, a more cost effective approach is to handle any potential issues through software.

Another problem that arises when assumptions are made that a network will always be reliable is that it makes it particularly difficult and expensive to guarantee reliable operations in WANs. WAN connections can be configured through multiple service providers that use independent paths with independent failure probabilities. However, reliable WAN links themselves tend to be expensive, and requiring two of them in many cases can be more than twice as expensive. Note also that many applications may not otherwise require that WAN links be reliable or redundant (for instance there may already be local replication and a temporary outage of cross-site replication may be acceptable for the application). Rather, the distributed system design mandates the requirement because without a reliable network, there is the potential for the system to become confused and corrupt data. Finally, many systems initially are not intended to be deployed in WANs but future requirements arise (expectedly or unexpectedly) which create the need for a WAN deployment. Accordingly, designing a system from the beginning in such a way that it can be readily extended to WAN's in the future is often a wise decision.

FT*m* makes no assumptions or requirements about network behavior. The correctness of the protocols and the integrity of data is guaranteed in all cases [1]. In the worst case, massive loss of network connectivity may cause FT*m*-based systems to temporarily reduce the services they provide, but the data will never be corrupted and the service will continue when connectivity is restored. For instance, the Echo data replication product will not allow new changes to the data to be made if there are not enough machines able to communicate to provide a "quorum." Even in extreme network failures, data integrity is never in question and the system will always resume where it left off before the network failure occurred.

10. Design Flaws Relating to Network Behavior

It is counter-intuitive but true that systems which are tolerant of arbitrary network behavior are easier to test

[1] The exception to this is data corruption in the network undetected by checksumming or other such mechanisms. FT*m* supports optionally adding an extra level of checksumming in case the underlying checksums are thought to be inadequate.

than those that are not. This is because with a system that is known to break down under different kinds of network failures it is very difficult to determine whether problems that may arise were caused by, for instance, a bug in the software or were instead caused by a spurious network delay that the system was not designed to be tolerant of. Systems with a simple definition of correctness are typically easier to test and failures are more reliably classified as errors in either design, implementation, or network related problems. If a system is designed in which certain network behavior will cause a failure then the typical response to any error is to assume that such behavior caused the problem.

An example of this is a system known to be intolerant of spurious network delays or outages. When a failure occurs in such systems, it is usually impossible to either prove or disprove whether a network delay caused the failure or whether it was caused by some other problem with the system. With applications designed to have well defined behavior under arbitrary network conditions, you cannot dismiss problems as resulting from a "bad" or "dirty" network; other reasons need to be found for anomalous behavior.

FT*m* solves this issue by enabling applications to tolerate arbitrary network conditions (ie. network behavior should never cause the application to break down), thus it becomes much easier to define appropriate behavior and to categorize problems when and as they are detected.

11. Adding Fault-Tolerance Late in the Development Process

A common mistake in designing distributed systems is to add fault-tolerant features at the end of the development process. This is typically because the demands of developing the non-fault-tolerant portion of the system are challenging enough that separating out fault-tolerance as a "feature" to add later becomes an attractive proposition. It is usually proposed in such a way as to make it sound like a natural approach to design and development: "We will begin with single node systems, add 2-node fault tolerance, and then 4 and 8-node fault tolerance in succeeding releases."

The problem is that fault-tolerance in distributed systems poses different and more complex types of issues than in all other kinds of programming. Most complex distributed systems do not have a smooth transition from single-node to multi-node designs. Accordingly, a large number of design issues need to be taken into account as early as possible. The later in the development process that these are addressed, the more difficult it becomes to avoid the pitfalls described in this document.

FT*m* addresses this by making fault-tolerance easy to add during the early stages of design and development. Having said this, FT*m*'s simple API's allow for it to be adopted even late in the development process. Nevertheless, the earlier fault tolerance is designed into a system the better.

Conclusion

Each of the failure scenarios and/or mistakes described in this document commonly occur in contemporary distributed applications. Some of the pitfalls arise out of design decisions that may in some cases have made short term sense due to particular circumstances, but in general should have been avoided. Most of these mistakes, however, arise due to an overall lack of understanding of the complexities inherent in distributed systems. The ultimate lesson here is that there are techniques which should be followed when developing fault-tolerant distributed systems so as to ensure that they operate correctly without making any assumptions or having any requirements about the behavior of the network. Use of these techniques will result in more flexible and less error prone systems.

Contacts

For information regarding Ventura Networks suite of Products, please contact sales@venturanetworksinc.com or visit www.venturanetworksinc.com.

For further technical information regarding this document, please contact Mark Hayden at mark@venturanetworksinc.com.