Fault Tolerance

Dealing with an imperfect world

Paul Krzyzanowski

April 2009

Introduction

If we look at the words *fault* and *tolerance*, we can define the *fault* as a malfunction or deviation from expected behavior and *tolerance* as the capacity for enduring or putting up with something. Putting the words together, fault tolerance refers to a system's ability to deal with malfunctions.

Faults

A fault in a system is some deviation from the expected behavior of the system: a malfunction. Faults may be due to a variety of factors, including hardware failure, software bugs, operator (user) error, and network problems.

Faults can be classified into one of three categories:

Transient faults

These occur once and then disappear. For example, a network message doesn't reach its destination but does when the message is retransmitted.

Intermittent faults

Intermittent faults are characterized by a fault occurring, then vanishing again, then reoccurring, then vanishing, ... These can be the most annoying of component faults. A loose connection is an example of this kind of fault.

Permanent faults

This type of failure is persistent: it continues to exist until the faulty component is repaired or replaced. Examples of this fault are disk head crashes, software bugs, and burnt-out power supplies.

Any of these faults may be either a fail-silent failure (also known as a fail-stop) or a Byzantine failure. A fail-silent fault is one where the faulty unit stops functioning and produces no bad output. More precisely, it either produces no output or produces output that clearly indicates that the component has failed. A Byzantine fault is one where the faulty unit continues to run but produces incorrect results. Dealing with Byzantine faults is obviously more troublesome.

[The term *Byzantine fault* is inspired by the Byzantine Generals Problem]

When we discuss fault tolerance, the familiar terms *synchronous* and *asynchronous* take on different meanings. By a synchronous system, we refer to one that responds to a message within a known, finite amount of time. An asynchronous system does not. Communicating via a serial port is an example of a synchronous system. Communicating via IP packets is an example of an asynchronous system.

Approaches to faults

We can try to design systems that minimize the presence of faults. Fault avoidance is a process where we go through design and validation steps to ensure that the system avoids being faulty in the first place. This can include formal validation, code inspection, testing, and using robust hardware.

Fault removal is an *ex post facto* approach where faults were encountered in the system and we managed to remove those faults. This could have been done through through testing, debugging, and verification as well as replacing failed components with better ones, adding heat sinks to fix thermal dissipation problems, et cetera.

Fault tolerance is the realization that we will always have faults (or the potential for faults) in our system and that we have to design the system in such a way that it will be tolerant of those faults. That is, the system should compensate for the faults and continue to function.

Achieving fault tolerance

The general approach to building fault tolerant systems is redundancy. Redundancy may be applied at several levels.

Information redundancy seeks to provide fault tolerance through replicating or coding the data. For example, a Hamming code can provide extra bits in data to recover a certain ratio of failed bits. Sample uses of information redundancy are parity memory, ECC (Error Correcting Codes) memory, and ECC codes on data blocks.

Time redundancy achieves fault tolerance by performing an operation several times. Timeouts and retransmissions in reliable point-to-point and group communication are examples of time redundancy. This form of redundancy is useful in the presence of transient or intermittent faults. It is of no use with permanent faults. An example is TCP/IP's retransmission of packets.

Physical redundancy deals with devices, not data. We add extra equipment to enable the system to tolerate the loss of some failed components. RAID disks and backup name servers are examples of physical redundancy.

When addressing physical redundancy, we can differentiate *redundancy* from *replication*. With replication, we have several units operating concurrently and a voting (quorum) system to select the outcome. With redundancy, only one unit is functioning while the redundant units are standing by to fill in in case the unit ceases to work.

Levels of availability

In designing a fault-tolerant system, we must realize that 100% fault tolerance can never be achieved. Moreover, the closer we try to get to 100%, the more costly our system will be.

To design a practical system, one must consider the degree of replication needed. This will be obtained from a statistical analysis for probable acceptable behavior. Factors that enter into this analysis are the average worst-case performance in a system without faults and the average worst-case performance in a system with faults.

Availability refers to the mount of time that a system is functioning ("available"). It is typically expressed as a percentage that refers to the fraction of time that the system is available to users. A system that is available 99.999% of the time (referred to as "five nines") will, on average, experience at most 5.26 minutes of downtime per year. This includes planned (hardware and software upgrades) and unplanned (network outages, hardware failures, fires, power outages, earthquakes) downtime.

Five nines is the classic standard of availability for telephony. Achieving it entails intensive software testing, redundant processors, backup generators, and earthquake-resilient installation. If all that ever happens you your system is that you lose power for a day once a year then your reliability is at 99.7% ("two nines"). You can compute an availability percentage by dividing the minutes of uptime by the minutes in a year (or hours of uptime by hours in a year, or anything similar). For example, if a system is expected to be down for three hours a year on average then the uptime percentage is 1-(180 minutes / 525600 minutes) = 99.97%. The following table shows some availability levels, their common terms, and the corresponding annual downtime.

Class	**Availability**	**Annual Downtime**
Continuous	100%	0
Fault Tolerant	99.999%	5 minutes
Fault Resilient	99.99%	53 minutes
High Availability	99.9%	8.3 hours
Normal Availability	99 - 99.5%	44-87 hours

How much redundancy?

A system is said to be *k*-fault tolerant if it can withstand *k* faults. If the components fail silently, then it is sufficient to have *k*+1 components to achieve *k* fault tolerance: *k* components can fail and one will still be working. This, of course, refers to each individual component — a single point of failure. For example, three power supplies will be 2-fault tolerant: two power supplies can fail and the system will still function.

If the components exhibit Byzantine faults, then a minimum of 2*k*+1 components are needed to achieve *k* fault tolerance. This provides a sufficient number of working components that will allow the good data to out-vote the bad data that is produced by the Byzantine faults. In the worst case, *k* components will fail (generating false results) but *k*+1 components will remain working properly, providing a majority vote that is correct.

Active replication

Active replication is a technique for achieving fault tolerance through physical redundancy. A common instantiation of this is triple modular redundancy (TMR). This design handles 2-fault tolerance with fail-silent faults or 1-fault tolerance with Byzantine faults.

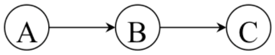

Figure 1. No redundancy

Under this system, we provide threefold replication of a component to detect and correct a single component failure. For example, consider a system where the output of A goes to the output of B and the output of B goes to C (Figure 1). Any single component failure will cause the entire system to fail.

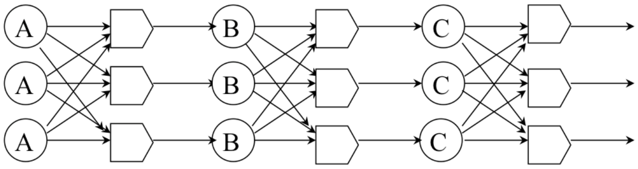

Figure 2. Triple Modular Redundancy (TMR)

In a TMR design, we replicate each component three ways and place voters after each stage to pick the majority outcome of the stage (Figure 2). The voter is responsible for picking the majority winner of the three inputs. A single Byzantine fault will be overruled by two good votes. The voters themselves are replicated because they too can malfunction.

In a software implementation, a client can replicate (or multicast) requests to each server. If requests are processed in order, all nonfaulty servers will yield the same replies. The requests must arrive reliably and in the same order on all servers. This requires the use of an atomic multicast.

Primary Backup (Active-Standby) approach

With a primary backup approach, one server (the *primary*) does all the work. When the server fails, the *backup* takes over.

To find out whether a primary failed, a backup may periodically ping the primary with *"are you alive"* messages. If it fails to get an acknowledgement, then the backup may assume that the primary failed and it will take over the functions of the primary. If the system is asynchronous, there are no upper bounds on a timeout value for the pings. This is a problem. Redundant networks can help ensure that a working communication channel exists. Another possible solution is to use a hardware mechanism to forcibly stop the primary.

This system is relatively easy to design since requests do not have to go be multicast to a group of machines and there are no decisions to be made on who takes over. An important point to note is that once the backup machine takes over another backup is needed immediately. Backup servers work poorly with Byzantine faults, since the backup may not be able to detect that the primary has actually failed.

Recovery from a primary failure may be time-consuming and/or complex depending on the needs for continuous operation and application recovery. Application failover is referred to by temperature grades. The easiest form of failover is known as cold failover. Cold failover entails application restart on the backup machine. When a backup machine takes over, it starts all the applications that were previously running on the primary system. Of course, any work that the primary may have done is now lost. With warm failover, applications periodically write checkpoint files onto stable storage that is shared with the backup system. When the backup system takes over, it reads the checkpoint files to bring the applications to the state of the last checkpoint. Finally, with hot failover, applications on the backup run in lockstep synchrony with applications on the primary, taking the same inputs as on the primary. When the backup takes over, it is in the exact state that the primary was in when it failed. However, if the failure was caused by software then there is a good chance that the backup died from the same bug since it received the same inputs.

Agreement in faulty systems

Distributed processes often have to agree on something. For example, they may have to elect a coordinator, commit a transaction, divide tasks, coordinate a critical section, etc. What happens when the processes and/or the communication lines are imperfect?

Two Army Problem

We'll first examine the case of good processors but faulty communication lines. This is known as the two army problem and can be summarized as follows:

Two divisions of an army, A and B, coordinate an attack on enemy army, C. A and B are physically separated and use a messenger to communicate. A sends a messenger to B with a message of "*let's attack at dawn*". B receives the message and agrees, sending back the messenger with an "*OK*" message. The messenger arrives at A, but A realizes that B did not know whether the messenger made it back safely. If B does is not convinced that A received the acknowledgement, then it will not be confident that the attack should take place since the army will not win on its own. A may choose to send the messenger back to B with a message of "*A received the OK*" but A will then be unsure as to whether B received this message. This is also known as the *multiple acknowledgment problem*. It demonstrates that even with non-faulty processors, ultimate agreement between two processes is *not* possible with unreliable communication. The best we can do is hope that it usually works.

Byzantine Generals Problem

> Solutions to the Byzantine Generals problem are *not* obvious, intuitive, or simple. They are not presented in these notes. You can read Lamport's paper on the problem here. You can also check out the brief summary and various solutions – which go beyond the Lamport paper – here.

The other case to consider is that of reliable communication lines but faulty processors. This is known as the Byzantine Generals Problem. In this problem, there are *n* army generals who head different divisions. Communication is reliable (radio or telephone) but *m* of the generals are traitors (faulty) and are trying to prevent others from reaching agreement by feeding them incorrect information. The question is: can the loyal generals still reach agreement? Specifically, each general knows the size of his division. At the end of the algorithm can each general know the troop strength of every other loyal division? Lamport demonstrated a solution that works for certain cases. His answer to this problem is that any solution to the problem of overcoming *m* traitors requires a minimum of 3*m*+1 participants (2*m*+1 loyal generals). This means that more than 2/3 of the generals must be loyal. Moreover, it was demonstrated that no protocol can overcome *m* faults with fewer than *m*+1 rounds of message exchanges and O(mn^2) messages. Clearly, this is a rather costly solution. While the Byzantine model may be applicable to certain types of special-purpose hardware, it will rarely be useful in general purpose distributed computing environments.

Examples of fault tolerance

ECC (Error Correction Code) memory

ECC memory contains extra logic that implements Hamming codes to detect and correct bit errors that are caused by fabrication errors, electrical disturbances, or neutron and cosmic ray radiation. A simple form of error detection is a parity code: one extra bit indicates whether the word has an odd or even number of bits. A single bit error will cause the parity to report an incorrect value and hence indicate an error. Most implementations of ECC memory use a Hamming code that detects two bit errors and corrects any single bit error per 64-bit word. In the general case, Hamming codes can be created for any number of bits (see the wikipedia article for a description of the algorithm).

This is an example of information redundancy. Information coding is used to provide fault tolerance for the data in memory (and, yes, the coding requires additional memory).

Machine failover via DNS SRV records

The fault tolerance goal here is to allow a client to connect to one functioning machine that represents a given hostname. Some machines may be inaccessible because they are out of service or the network connection to them is not functioning.

Instead of using DNS (Domain Name System) to resolve a hostname to one IP address, the client will use DNS to look up SRV records for that name. The SRV record is a somewhat generic record in DNS that allows one to specify information on available services for a host (see RFC 2782). Each SRV record contains a priority, weight, port, and target hostname. The priority field is used to prioritize the list of servers. An additional weight value can then be used to balance the choice of several servers of equal priority. Once a server is selected, DNS is used to look up the address of the target hostname field. If that server doesn't respond then the client tries another machine in the list.

This approach is commonly used in voice over IP (VoIP) systems to pick a SIP server (or SIP proxy) among several available SIP servers for a specific hostname. DNS MX records (mail servers for a hostname) take the same approach: use DNS to look up the list of mail servers for a host and then connect to one that works.

This is an example of physical redundancy.

RAID 1 (disk mirroring)

RAID stands for Redundant Array of Independent Disks (it used to stand for *Redundant Array of Inexpensive Disks* but that did not make for a good business model selling expensive RAID systems). RAID supports different configurations, known as levels. RAID 0, for example, refers to disk striping, where data is spread out across two disks. For example, disk 0 holds blocks 0, 2, 4, 6, ... and disk 1 holds blocks 1, 3, 5, 7, ... This level offers no fault tolerance and is designed to provide higher performance: two blocks can often be written or read concurrently.

RAID 1 is a disk mirroring configuration. All data that is written to one disk is also written to a second disk. A block of data can be read from either disk. If one disk goes out of service, the remaining disk will have all the data.

RAID 1 is an example of an active-active approach to of physical redundancy. As opposed to the primary-server (active-passive) approach, both systems are in use at all times.

RAID 4/RAID 5 (disk parity)

RAID 4 and RAID 5 use block-level striping together with parity to provide 1-fault tolerance. A *stripe* refers to a set of blocks that is spread out across a set of *n* disks, with one block per disk. A parity block is written to disk *n*+1. The parity is the exclusive-or of the set of blocks in each stripe. If one disk fails, its contents are recovered by computing an exclusive-or of all the blocks in that stripe set together with the parity block.

RAID 5 is the exact same thing but the parity blocks are distributed among all the disks so that writing parity does not become a bottleneck. For example, in a four disk configuration, the parity block may be on disk 0 for the first stripe, disk 1 for the second stripe, disk 2, for the third stripe, disk 3 for the fourth stripe, disk 0 for the fifth stripe, etc.

RAID 4 and RAID 5 are examples of information redundancy. As with ECC, we need additional physical hardware but we are achieving the fault tolerance through information coding, not by having a standby disk that contains a replica of the data.

TCP retransmission

TCP/IP (Transmission Control Protocol) is the reliable virtual circuit service transport layer protocol provided on top of the unreliable network layer Internet Protocol. TCP relies that the sender receives an acknowledgement from the receiver whenever a packet is received. If the sender does not receive that acknowledgment within a certain amount of time, it assumes that the packet was lost. The sender then retransmits the packet. In Windows, the retransmission timer is initialized to three seconds. The default maximum number of retransmissions is five. The retransmission time is adjusted based on the usual delay of a specific connection. (See RFC 2581, TCP Congestion Control.)

TCP is an example of time redundancy