Tandem TR 85.7

WHY DO COMPUTERS STOP AND WHAT CAN BE DONE ABOUT IT?

Jim Gray

June, 1985

Revised November, 1985

ABSTRACT

An analysis of the failure statistics of a commercially available fault-tolerant system shows that administration and software are the major contributors to failure. Various approaches to software fault-tolerance are then discussed -- notably process-pairs, transactions and reliable storage. It is pointed out that faults in production software are often soft (transient) and that a transaction mechanism combined with persistent process-pairs provides fault-tolerant execution -- the key to software fault-tolerance.

DISCLAIMER

This paper is not an "official" Tandem statement on fault-tolerance. Rather, it expresses the author's research on the topic.

An early version of this paper appeared in the proceedings of the German Association for Computing Machinery Conference on Office Automation, Erlangen, Oct. 2-4, 1985.

TABLE OF CONTENTS

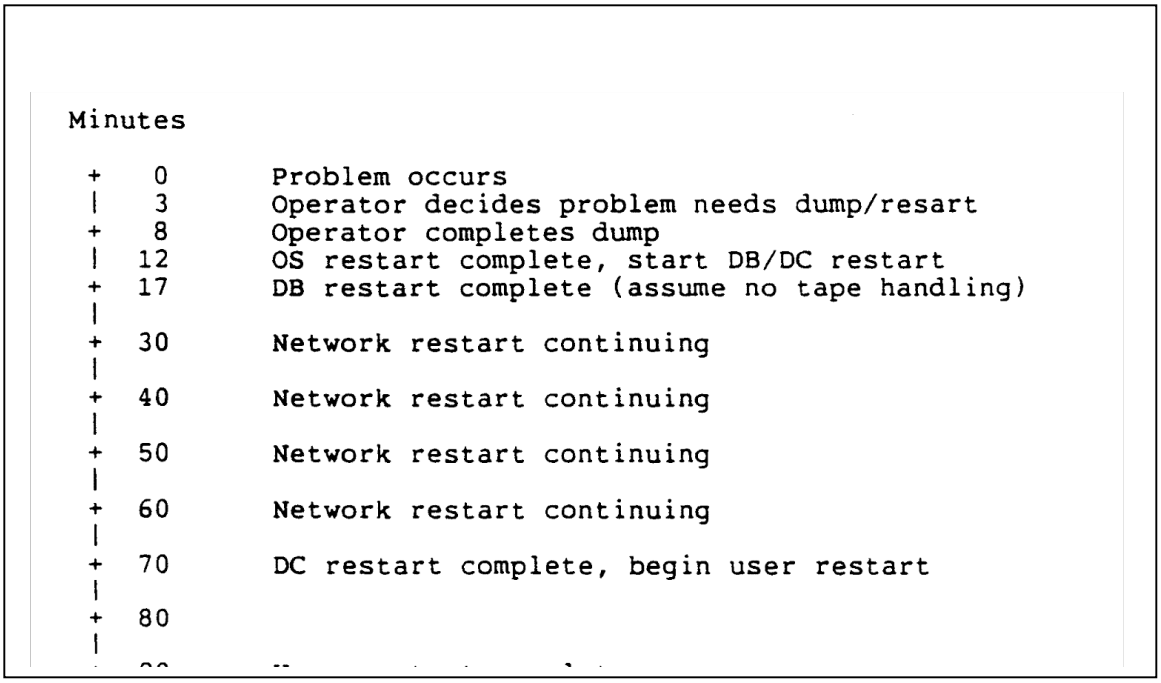
Introduction

Computer applications such as patient monitoring, process control, online transaction processing, and electronic mail require high availability.

The anatomy of a typical large system failure is interesting: Assuming, as is usually the case, that an operations or software fault caused the outage, Figure 1 shows a time line of the outage. It takes a few minutes for someone to realize that there is a problem and that a restart is the only obvious solution. It takes the operator about 5 minutes to snapshoot the system state for later analysis. Then the restart can begin. For a large system, the operating system takes a few minutes to get started. Then the database and data communications systems begin their restart. The database restart completes within a few minutes but it may take an hour to restart a large terminal network. Once the network is up, the users take a while to refocus on the tasks they had been performing. After restart, much work has been saved for the system to perform -- so the transient load presented at restart is the peak load. This affects system sizing.

Conventional well-managed transaction processing systems fail about once every two weeks [Mourad], [Burman]. The ninety minute outage outlined above translates to 99.6% availability for such systems. 99.6% availability "sounds" wonderful, but hospital patients, steel mills, and electronic mail users do not share this view -- a 1.5 hour outage every ten days is unacceptable. Especially since outages usually come at times of peak demand [Mourad].

These applications require systems which virtually never fail -- parts of the system may fail but the rest of the system must tolerate failures and continue delivering service. This paper reports on the structure and success of such a system -- the Tandem NonStop system. It has *MTBF* measured in years -- more than two orders of magnitude better than conventional designs.


Minutes
0 Problem occurs
3 Operator decides problem needs dump/resart
8 Operator completes dump
12 OS restart complete, start DB/DC restart
17 DB restart complete (assume no tape handling)
30 Network restart continuing
40 Network restart continuing
50 Network restart continuing
60 Network restart continuing
70 DC restart complete, begin user restart
80

Hardware Availability Modular Redundancy

Reliability and availability are different: Availability is doing the right thing within the specified response time. Reliability is not doing the wrong thing.

Expected reliability is proportional to the Mean Time Between Failures (*MTBF*). A failure has some Mean Time To Repair (*MTTR*). Availability can be expressed as a probability that the system will be available:

$$Availability \approx \frac{MTBF}{MTBF + MTTR}$$

In distributed systems, some parts may be available while others are not. In these situations, one weights the availability of all the devices (e.g. if 90% of the database is available to 90% of the terminals, then the system is .9x.9 = 81% available.)

The key to providing high availability is to modularize the system so that modules are the unit of failure and replacement. Spare modules are configured to give the appearance of instantaneous repair -- if *MTTR* is tiny, then the failure is "seen" as a delay rather than a failure. For example, geographically distributed terminal networks frequently have one terminal in a hundred broken. Hence, the system is limited to 99% availability (because terminal availability is 99%). Since terminal and communications line failures are largely independent, one can provide very good "site" availability by placing two terminals with two communications lines at each site. In essence, the second ATM provides instantaneous repair and hence very high availability. Moreover, they increase transaction throughput at locations with heavy traffic. This approach is taken by several high availability Automated Teller Machine (ATM) networks.

This example demonstrates the concept: modularity and redundancy allows one module of the system to fail without affecting the availability of the system as a whole because redundancy leads to small *MTTR*. This combination of modularity and redundancy is the key to providing continuous service even if some components fail.

Von Neumann was the first to analytically study the use of redundancy to construct available (highly reliable) systems from unreliable components [Neumann]. In his model, a redundancy 20,000 was needed to get a system *MTBF* of 100 years. Certainly, his components were less reliable than transistors, he was thinking of human neurons or vacuum tubes. Still, it is not obvious why von Neumann's machines required a redundancy factor of 20,000 while current electronic systems use a factor of 2 to achieve very high availability. The key difference is that von Neumann's model lacked modularity, a failure in any bundle of wires anywhere, implied a total system failure.

VonNeumann's model had redundancy without modularity. In contrast, modern computer systems are constructed in a modular fashion -- a failure within a module only affects that module. In addition each module is constructed to be fail-fast -- the module either functions properly or stops [Schlichting]. Combining redundancy with modularity allows one to use a redundancy of two rather than 20,000. Quite an economy!

To give an example, modern discs are rated for an *MTBF* above 10,000 hours -- a hard fault once a year. Many systems duplex pairs of such discs, storing the same information on both of them, and using independent paths and controllers for the discs. Postulating a very leisurely *MTTR* of 24 hours and assuming independent failure modes, the *MTBF* of this pair (the mean time to a double failure within a 24 hour window) is over 1000 years. In practice, failures are not quite independent, but the *MTTR* is less than 24 hours and so one observes such high availability.

Generalizing this discussion, fault-tolerant hardware can be constructed as follows:

- Hierarchically decompose the system into modules.
- Design the modules to have *MTBF* in excess of a year.
- Make each module fail-fast -- either it does the right thing or stops.
- Detect module faults promptly by having the module signal failure or by requiring it to periodically send an *I AM ALIVE* message or reset a watchdog timer.
- Configure extra modules which can pick up the load of failed modules. Takeover time, including the detection of the module failure, should be seconds. This gives an apparent module *MTBF* measured in millennia.

The resulting systems have hardware *MTBF* measured in decades or centuries.

This gives fault-tolerant hardware. Unfortunately, it says nothing about tolerating the major sources of failure: software and operations. Later we show how these same ideas can be applied to gain software fault-tolerance.

An Analysis of Failures of a Fault-Tolerant System

There have been many studies of why computer systems fail. To my knowledge, none have focused on a commercial fault-tolerant system. The statistics for fault-tolerant systems are quite a bit different from those for conventional mainframes [Mourad]. Briefly, the *MTBF* of hardware, software and operations is more than 500 times higher than those reported for conventional computing systems -- fault-tolerance works. On the other hand, the ratios among the sources of failure are about the same as those for conventional systems. Administration and software dominate; hardware and environment are minor contributors to total system outages.

Tandem Computers Inc. makes a line of fault-tolerant systems [Bartlett] [Borr 81, 84]. I analyzed the causes of system failures reported to Tandem over a seven-month period. The sample set covered more than 2000 systems and represents over 10,000,000 system hours or over 1300 system years. Based on interviews with a sample of customers, I believe these reports cover about 50% of all total system failures. There is under-reporting of failures caused by customers or by environment. Almost all failures caused by the vendor are reported.

During the measured period, 166 failures were reported including one fire and one flood. Overall, this gives a system *MTBF* of 7.8 years reported and 3.8 years *MTBF* if the systematic under-reporting is taken into consideration. This is still well above the 1 week *MTBF* typical of conventional designs.

By interviewing four large customers who keep careful books on system outages, I got a more accurate picture of their operation. They averaged a 4-year *MTBF* (consistent with 7.8 years with 50% reporting). In addition, their failure statistics had under-reporting in the expected areas of environment and operations. Rather than skew the data by multiplying all *MTBF* numbers by .5, I will present the analysis as though the reports were accurate.

About one third of the failures were “infant mortality” failures -- a product having a recurring problem. All these fault clusters are related to a new software or hardware product still having the bugs shaken out. If one subtracts out systems having “infant” failures or non-duplexed-disc failures, then the remaining failures, 107 in all, make an interesting analysis (see table 1).

First, the system *MTBF* rises from 7.8 years to over 11 years.

System administration, which includes operator actions, system configuration, and system maintenance, was the main source of failures -- 42%. Software and hardware maintenance was the largest category. High availability systems allow users to add software and hardware and to do preventative maintenance while the system is operating. By and large, online maintenance works VERY well. It extends system availability by two orders of magnitude. But occasionally, once every 52 years by my figures, something goes wrong. This number is somewhat speculative -- if a system failed while it was undergoing online maintenance or while hardware or software was being added, I ascribed the failure to maintenance. Sometimes it was clear that the maintenance person typed the wrong command or unplugged the wrong module, thereby introducing a double failure. Usually, the evidence was circumstantial. The notion that mere humans make a single critical mistake every few decades amazed me -- clearly these people are very careful and the design tolerates some human faults.

System Failure Mode	Probability		MTBF in years
Administration	42%		31 years
Maintenance:	25%		
Operations	9%	(?)	
Configuration	8%		
Software	25%		50 years
Application	4%	(?)	
Vendor	21%		
Hardware	18%		73 years
Central	1%		
Disc	7%		
Tape	2%		
Comm Controllers	6%		
Power supply	2%		
Environment	14%		87 years
Power	9%	(?)	
Communications	3%		
Facilities	2%		
Unknown	3%		
Total	103%		11 years

Table 1. Contributors to Tandem System outages reported to the vendor. As explained in the text, infant failures (30%) are subtracted from this sample set. Items marked by "?" are probably under-reported because the customer does not generally complain to the vendor about them. Power outages below 4 hours are tolerated by the NonStop system and hence are under-reported. We estimate 50% total under-reporting.

System operators were a second source of human failures. I suspect under-reporting of these failures. If a system fails because of the operator, he is less likely to tell us about it. Even so, operators reported several failures. System configuration, getting the right collection of software, microcode, and hardware, is a third major headache for reliable system administration.

Software faults were a major source of system outages -- 25% in all. Tandem supplies about 4 million lines of code to the customer. Despite careful efforts, bugs are present in this software. In addition, customers write quite a bit of software. Application software faults are probably under-reported here. I guess that only 30% are reported. If that is true, application programs contribute 12% to outages and software rises to 30% of the total.

Next come environmental failures. Total communications failures (losing all lines to the local exchange) happened three times; in addition, there was a fire and a flood. No outages caused by cooling or air conditioning were reported. Power outages are a major source of failures among customers who do not have emergency backup power (North American urban power typically has a 2-month *MTBF*). Tandem systems tolerate over 4 hours of lost power without losing any data or communications state (the *MTTR* is almost zero), so customers do not generally report minor power outages (less than 1 hour) to us.

Given that power outages are under-reported, the smallest contributor to system outages was hardware, mostly discs and communications controllers. The measured set included over 20,000 discs -- over 100,000,000 disc hours. We saw 19 duplexed disc failures, but if one subtracts out the infant mortality failures, then there were only 7 duplexed disc failures. In either case, one gets an MTBF in excess of 5 million hours for the duplexed pair and their controllers. This approximates the 1000-year *MTBF* calculated in the earlier section.

Implications of the Analysis of *MTBF*

The implications of these statistics are clear: the key to high-availability is tolerating operations and software faults.

Commercial fault-tolerant systems are measured to have a 73-year hardware *MTBF* (Table 1). I believe there was 75% reporting of outages caused by hardware. Calculating from device *MTBF*, there were about 50,000 hardware faults in the sample set. Less than one in a thousand resulted in a double failure or an interruption of service. Hardware fault-tolerance works!

In the future, hardware will be even more reliable due to better design, increased levels of integration, and reduced numbers of connectors.

By contrast, the trend for software and system administration is not positive. Systems are getting more complex. In this study, administrators reported 41 critical mistakes in over 1300 years of operation. This gives an operations *MTBF* of 31 years! Operators certainly made many more mistakes, but most were not fatal. These administrators are clearly very careful and use good practices.

The top priority for improving system availability is to reduce administrative mistakes by making self-configured systems with minimal maintenance and minimal operator interaction. Interfaces that ask the operator for information or ask him to perform some function must be simple, consistent and operator fault-tolerant.

The same discussion applies to system maintenance. Maintenance interfaces must be simplified. Installation of new equipment must have fault-tolerant procedures and the maintenance interfaces must be simplified or eliminated. To give a concrete example, Tandem's newest discs have no special customer engineering training (installation is "obvious") and they have no scheduled maintenance.

A secondary implication of the statistics is actually a contradiction:

- New and changing systems have higher failure rates. Infant products contributed one third of all outages. Maintenance caused one third of the remaining outages. A way to improve availability is to install proven hardware, and software, and then leave it alone. As the adage says, "If it's not broken, don't fix it".

- On the other hand, a Tandem study found that a high percentage of outages were caused by "known" hardware or software bugs, which had fixes available, but the fixes were not yet installed in the failing system. This suggests that one should install software and hardware fixes as soon as possible.

There is a contradiction here: never change it and change it ASAP! By consensus, the risk of change is too great. Most installations are slow to install changes; they rely on fault-tolerance to protect them until the next major release. After all, it worked yesterday, so it will probably work tomorrow.

Here one must separate software and hardware maintenance. Software fixes outnumber hardware fixes by several orders of magnitude. I believe this causes the difference in strategy between hardware and software maintenance. One cannot forego hardware preventative maintenance -- our studies show that it may be good in the short term but it is disastrous in the long term. One must install hardware fixes in a timely fashion. If possible, preventative maintenance should be scheduled to minimize the impact of a possible mistake. Software appears to be different. The same study recommends installing a software fix only if the bug is causing outages. Otherwise, the study recommends waiting for a major software release, and carefully testing it in the target environment prior to installation. Adams comes to similar conclusions [Adams]; he points out that for most bugs, the chance of "rediscovery" is very slim indeed.

The statistics also suggest that if availability is a major goal, then avoid products which are immature and still suffering infant mortality. It is fine to be on the leading edge of technology, but avoid the bleeding edge of technology.

The last implication of the statistics is that software fault-tolerance is important. Software fault-tolerance is the topic of the rest of the paper.

Fault-tolerant Execution

Based on the analysis above, software accounts for over 25% of system outages. This is quite good -- a *MTBF* of 50 years! The volume of Tandem's software is 4 million lines and growing at about 20% per year. Work continues on improving coding practices and code testing but there is little hope of getting ALL the bugs out of all the software. Conservatively, I guess one bug per thousand lines of code remains after a program goes through design reviews, quality assurance, and beta testing. That suggests the system has several thousand bugs. But somehow, these bugs cause very few system failures because the system tolerates software faults.

The keys to this software fault-tolerance are:

- Software modularity through processes and messages.
- Fault containment through fail-fast software modules.
- Process-pairs to tolerate hardware and transient software faults.
- Transaction mechanism to provide data and message integrity.
- Transaction mechanism combined with process-pairs to ease exception handling and tolerate software faults.

This section expands on each of these points.

- **Software modularity through processes and messages**

As with hardware, the key to software fault-tolerance is to hierarchically decompose large systems into modules, each module being a unit of service and a unit of failure. A failure of a module does not propagate beyond the module.

There is considerable controversy about how to modularize software. Starting with Burroughs' Esbol and continuing through languages like Mesa and Ada, compiler writers have assumed perfect hardware and contended that they can provide good fault isolation through static compile-time type checking. In contrast, operating systems designers have advocated run-time checking combined with the process as the unit of protection and failure.

Although compiler checking and exception handling provided by programming languages are real assets, history seems to have favored the run-time checks plus the

process approach to fault containment. It has the virtue of simplicity -- if a process or its processor misbehaves, stop it. The process provides a clean unit of modularity, service, fault containment, and failure.

- **Fault containment through fail-fast software modules**

The process approach to fault isolation advocates that the process software module be fail-fast, it should either function correctly or it should detect the fault, signal failure and stop operating.

Processes are made fail-fast by defensive programming. They check all their inputs, intermediate results, outputs and data structures as a matter of course. If any error is detected, they signal a failure and stop. In the terminology of [Christian], fail-fast software has small fault detection latency.

The process achieves fault containment by sharing no state with other processes; rather, its only contact with other processes is via messages carried by a kernel message system.

- **Software faults are soft -- the Bohrbug/Heisenbug hypothesis**

Before developing the next step in fault-tolerance, process-pairs, we need to have a software failure model. It is well known that most hardware faults are soft -- that is, most hardware faults are transient. Memory error correction and checksums plus retransmission for communication are standard ways of dealing with transient hardware faults. These techniques are variously estimated to boost hardware *MTBF* by a factor of 5 to 100.

I conjecture that there is a similar phenomenon in software -- most production software faults are soft. If the program state is reinitialized and the failed operation retried, the operation will usually not fail the second time.

If you consider an industrial software system that has gone through structured design, design reviews, quality assurance, alpha test, beta test, and months or years of production, then most of the "hard" software bugs, ones that always fail on retry, are gone. The residual bugs are rare cases, typically related to strange hardware conditions (rare or transient device fault), limit conditions (out of storage, counter overflow, lost interrupt, etc.) or race conditions (forgetting to request a semaphore).

In these cases, resetting the program to a quiescent state and re-executing it will quite likely work, because now the environment is slightly different. After all, it worked a minute ago!

The assertion that most production software bugs are soft -- Heisenbugs that go away when you look at them -- is well known to systems programmers. Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring. But Heisenbugs may elude a bugcatcher for years of execution. Indeed, the bugcatcher may perturb the situation just enough to make the Heisenbug disappear. This is analogous to the Heisenberg Uncertainty Principle in Physics.

I have tried to quantify the chances of tolerating a Heisenbug by re-execution. This is difficult. A poll yields nothing quantitative. The one experiment I did went as follows: The spooler error log of several dozen systems was examined. The spooler is constructed as a collection of fail-fast processes. When one of the processes detects a fault, it stops and lets its brother continue the operation. The brother does a software retry. If the brother also fails, then the bug is a Bohrbug rather than a Heisenbug. In the measured period, one out of 132 software faults was a Bohrbug, the remainders were Heisenbugs.

A related study is reported in [Mourad]. In MVS/XA functional recovery routines try to recover from software and hardware faults. If a software fault is recoverable, it is a Heisenbug. In that study, about 90% of the software faults in system software had functional recovery routines (FRRs). Those routines had a 76% success rate in continuing system execution. That is, MVS FRRs extend the system software *MTBF* by a factor of 4.

It would be nice to quantify this phenomenon further. As it is, systems designers know from experience that they can exploit the Heisenbug hypothesis to improve software fault-tolerance.

- **Process-pairs for fault-tolerant execution**

One might think that fail-fast modules would produce a reliable but unavailable system -- modules are stopping all the time. But, as with fault-tolerant hardware, configuring extra software modules gives a *MTTR* of milliseconds in case a process fails due to a hardware failure or a software Heisenbug. If modules have a *MTBF* of a year, then dual processes give very acceptable *MTBF* for the pair. Process triples do not improve *MTBF* because other parts of the system (e.g., operators) have orders of magnitude worse *MTBF*. So, in practice fault-tolerant processes are generically called process-pairs. There are several approaches to designing process-pairs:

Lockstep: In this design, the primary and backup processes synchronously execute the same instruction stream on independent processors [Kim]. If one of the processors fails, the other simply continues the computation. This approach gives good tolerance to hardware failures but gives no tolerance of Heisenbugs. Both streams will execute any programming bug in lockstep and will fail in exactly the same way.

State Checkpointing: In this scheme, communication sessions are used to connect a requestor to a process-pair. The primary process in a pair does the computation and sends state changes and reply messages to its backup prior each major event. If the primary process stops, the session switches to the backup process which continues the conversation with the requestor. Session sequence numbers are used to detect duplicate and lost messages, and to resend the reply if a duplicate request arrives [Bartlett]. Experience shows that checkpointing process-pairs give excellent fault-tolerance (see Table 1), but that programming checkpoints is difficult. The trend is away from this approach and towards the Delta or Persistent approaches described below.

Automatic Checkpointing: This scheme is much like state checkpoints except that the kernel automatically manages the checkpointing, relieving the programmer of this chore. As described in [Borg], all messages to and from a process are saved by the message kernel for the backup process. At takeover, these messages are replayed to the backup to roll it forward to the primary process' state. When substantial computation or storage is required in the backup, the primary state is copied to the backup so that the message log and replay can be discarded. This scheme seems to send more data than the state checkpointing scheme and hence seems to have high execution cost.

Delta Checkpointing: This is an evolution of state checkpointing. Logical rather than physical updates are sent to the backup [Borr 84]. Adoption of this scheme by

Tandem cut message traffic in half and message bytes by a factor of 3 overall [Enright]. Deltas have the virtue of performance as well as making the coupling between the primary and backup state logical rather than physical. This means that a bug in the primary process is less likely to corrupt the backup's state.

Persistence: In persistent process-pairs, if the primary process fails, the backup wakes up in the null state with amnesia about what was happening at the time of the primary failure. Only the opening and closing of sessions is checkpointed to the backup. These are called stable processes by [Lampson]. Persistent processes are the simplest to program and have low overhead. The only problem with persistent processes is that they do not hide failures! If the primary process fails, the database or devices it manages are left in a mess and the requestor notices that the backup process has amnesia. We need a simple way to resynchronize these processes to have a common state. As explained below, transactions provide such a resynchronization mechanism.

Summarizing the pros and cons of these approaches:

- Lockstep processes don't tolerate Heisenbugs.
- State checkpoints give fault-tolerance but are hard to program.
- Automatic checkpoints seem to be inefficient -- they send a lot of data to the backup.
- Delta checkpoints have good performance but are hard to program.
- Persistent processes lose state in case of failure.

We argue next that transactions combined with persistent processes are simple to program and give excellent fault-tolerance.

- **Transactions for data integrity**

A transaction is a group of operations, be they database updates, messages, or external actions of the computer, which form a consistent transformation of the state.

Transactions should have the ACID property [Haeder]:

- Atomicity: Either all or none of the actions of the transaction should "happen". Either it commits or aborts.
- Consistency: Each transaction should see a correct picture of the state, even if concurrent transactions are updating the state.
- Integrity: The transaction should be a correct state transformation.
- Durability: Once a transaction commits, all its effects must be preserved, even if there is a failure.

The programmer's interface to transactions is quite simple: he starts a transaction by asserting the BeginTransaction verb, and ends it by asserting the EndTransactjon or AbortTransaction verb. The system does the rest.

The classical implementation of transactions uses locks to guarantee consistency and a log or audit trail to insure atomicity and durability. Borr shows how this concept generalizes to a distributed fault-tolerant system [Borr 81, 84].

Transactions relieve the application programmer of handling many error conditions. If things get too complicated, the programmer (or the system) calls AbortTransaction which cleans up the state by resetting everything back to the beginning of the transaction.

- **Transactions for simple fault-tolerant execution**

Transactions provide reliable execution and data availability (recall reliability means not doing the wrong thing, availability means doing the right thing and on time). Transactions do not directly provide high system availability. If hardware fails or if there is a software fault, most transaction processing systems stop and go through a system restart -- the 90 minute outage described in the introduction.

It is possible to combine process-pairs and transactions to get fault-tolerant execution and hence avoid most such outages.

As argued above, process-pairs tolerate hardware faults and software Heisenbugs. But most kinds of process-pairs are difficult to implement. The "easy" process-pairs, persistent process-pairs, have amnesia when the primary fails and the backup takes over. Persistent process-pairs leave the network and the database in an unknown state when the backup takes over.

The key observation is that the transaction mechanism knows how to UNDO all the changes of incomplete transactions. So we can simply abort all uncommitted transactions associated with a failed persistent process and then restart these transactions from their input messages. This cleans up the database and system states, resetting them to the point at which the transaction began.

So, persistent process-pairs plus transactions give a simple execution model which continues execution even if there are hardware faults or Heisenbugs. This is the key to the Encompass data management system's fault-tolerance [Borr 81]. The programmer writes fail-fast modules in conventional languages (Cobol, Pascal, Fortran) and the transaction mechanism plus persistent process-pairs makes his program robust.

Unfortunately, people implementing the operating system kernel, the transaction mechanism itself and some device drivers still have to write "conventional" process-pairs, but application programmers do not. One reason Tandem has integrated the transaction mechanism with the operating system is to make the transaction mechanism available to as much software as possible [Borr 81].

Fault-tolerant Communication

Communications lines are the most unreliable part of a distributed computer system, partly because they are so numerous and partly because they have poor *MTBF*. The operations aspects of managing them, diagnosing failures and tracking the repair process are a real headache [Gray].

At the hardware level, fault-tolerant communication is obtained by having multiple data paths with independent failure modes.

At the software level, the concept of session is introduced. A session has simple semantics: a sequence of messages is sent via the session. If the communication path fails, an alternate path is tried. If all paths are lost, the session endpoints are told of the failure. Timeout and message sequence numbers are used to detect lost or duplicate messages. All this is transparent above the session layer.

Sessions are the thing that makes process-pairs work: the session switches to the backup of the process-pair when the primary process fails [Bartlett]. Session sequence numbers (called SyncIDs by Bartlett) resynchronize the communication state between the sender and receiver and make requests/replies idempotent.

Transactions interact with sessions as follows: if a transaction aborts, the session sequence number is logically reset to the sequence number at the beginning of the transaction and all intervening messages are canceled. If a transaction commits, the messages on the session will be reliably delivered EXACTLY once [Spector].

Fault-tolerant Storage

The basic form of fault-tolerant storage is replication of a file on two media with independent failure characteristics -- for example two different disc spindles or, better yet, a disc and a tape. If one file has an *MTBF* of a year then two files will have a millennia *MTBF* and three copies will have about the same *MTBF* -- as the Tandem system failure statistics show, other factors will dominate at that point.

Remote replication is an exception to this argument. If one can afford it, storing a replica in a remote location gives good improvements to availability. Remote replicas will have different administrators, different hardware, and different environment. Only the software will be the same. Based on the analysis in Table 1, this will protect against 75% of the failures (all the non-software failures). Since it also gives excellent protection against Heisenbugs, remote replication guards against most software faults.

There are many ways to remotely replicate data; one can have exact replicas, can have the updates to the replica done as soon as possible or even have periodic updates. [Gray] describes representative systems which took different approaches to long-haul replication.

Transactions provide the ACID properties for storage -- Atomicity, Consistency, Integrity and Durability [Haeder]. The transaction journal plus an archive copy of the data provide a replica of the data on media with independent failure modes. If the primary copy fails, a new copy can be reconstructed from the archive copy by applying all updates committed since the archive copy was made. This is Durability of data.

In addition, transactions coordinate a set of updates to the data, assuring that all or none of them apply. This allows one to correctly update complex data structures without concern for failures. The transaction mechanism will undo the changes if something goes wrong. This is Atomicity.

A third technique for fault-tolerant storage is partitioning the data among discs or nodes and hence limiting the scope of a failure. If the data is geographically partitioned, local users can access local data even if the communication net or remote nodes are down. Again, [Gray] gives examples of systems which partition data for better availability.

Summary

Computer systems fail for a variety of reasons. Large computer systems of conventional design fail once every few weeks due to software, operations mistakes, or hardware. Large fault-tolerant systems are measured to have an *MTBF* at orders of magnitude higher -- years rather than weeks.

The techniques for fault-tolerant hardware are well documented. They are quite successful. Even in a high availability system, hardware is a minor contributor to system outages.

By applying the concepts of fault-tolerant hardware to software construction, software *MTBF* can be raised by several orders of magnitude. These concepts include: modularity, defensive programming, process-pairs, and tolerating soft faults -- Heisenbugs.

Transactions plus persistent process-pairs give fault-tolerant execution. Transactions plus resumable communications sessions give fault-tolerant communications. Transactions plus data replication give fault-tolerant storage. In addition, transaction atomicity coordinates the changes of the database, the communications net, and the executing processes. This allows easy construction of high availability software.

Dealing with system configuration, operations, and maintenance remains an unsolved problem. Administration and maintenance people are doing a much better job than we have reason to expect. We can't hope for better people. The only hope is to simplify and reduce human intervention in these aspects of the system.

Acknowledgments

The following people helped in the analysis of the Tandem system failure statistics: Robert Bradley, Jim Enright, Cathy Fitzgerald, Sheryl Hainlin, Pat Helland, Dean Judd, Steve Logsdon, Franco Putzolu, Carl Niehaus, Harald Sammer, and Duane Wolfe. In presenting the analysis, I had to make several outrageous assumptions and "integrate" contradictory stories from different observers of the same events. For that, I must take full responsibility. Robert Bradley, Gary Gilbert, Bob Horst, Dave Kinkade, Carl Niehaus, Carol Minor, Franco Putzolu, and Bob White made several comments that clarified the presentation. Special thanks are due to Joel Bartlett and especially Flaviu Cristian who tried very hard to make me be more accurate and precise.

References

[Adams] Adams, E., "Optimizing Preventative Service of Software Products", IBM J. Res. and Dev., Vol. 28, No. 1, Jan. 1984.

[Bartlett] Bartlett, J.,"A Nonstop Kernel," Proceedings of the Eighth Symposium on Operating System Principles, pp. 22-29, Dec. 1981.

[Borg] Borg, A., Baumbach, J., Glazer, S., "A Message System Supporting Fault-tolerance", ACM OS Review, Vol. 17, No. 5, 1984.

[Borr 81] Borr, A., "Transaction Monitoring in ENCOMPASS," Proc. 7Th VLDB, September 1981. Also Tandem Computers TR 81.2.

[Borr 84] Borr, A., "Robustness to Crash in a Distributed Database: A Non Shared-Memory Multi-processor Approach," Proc. 9th VLDB, Sept. 1984. Also Tandem Computers TR 84.2.

[Burman] Burman, M. "Aspects of a High Volume Production Online Banking System", Proc. Int. Workshop on High Performance Transaction Systems, Asilomar, Sept. 1985.

[Cristian] Cristian, F., "Exception Handling and Software Fault Tolerance", IEEE Trans. on Computers, Vol. c-31, No. 6, 1982.

[Enright] Enright, J. "DP2 Performance Analysis", Tandem memo, 1985.

[Gray] Gray, J., Anderton, M., "Distributed Database Systems -- Four Case Studies", to appear in IEEE TODS, also Tandem TR 85.5.

[Haeder] Haeder, T., Reuter, A., "Principals of Transaction-Oriented Database Recovery", ACM Computing Surveys, Vol. 15, No. 4, Dec. 1983.

[Kim] Kim, W., "Highly Available Systems for Database Applications", ACM Computing Surveys, Vol. 16, No. 1, March 1984

[Lampson] Lampson, B.W. ed, Lecture Notes in Computer Science Vol. 106, Chapter 11, Springer Verlag, 1982.

[Mourad] Mourad, S. and Andrews, D., "The Reliability of the IBM/XA Operating System", Digest of 15th Annual Int. Sym. on Fault-Tolerant Computing, June 1985. IEEE Computer Society Press.

[Schlichting] Schlichting, R.D., Schneider, F.B., "Fail-Stop Processors, an Approach to Designing Fault-Tolerant Computing Systems", ACM TOCS, Vol. 1, No. 3, Aug. 1983.

[Spector] "Multiprocessing Architectures for Local Computer Networks", PhD Thesis, STAN-CS-81-874, Stanford 1981.

[von Neumann] von Neumann, J. "Probabilistic Logics and the Synthesis of Reliable Organisms From Unreliable Components", Automata Studies, Princeton University Press, 1956.