Fault-tolerant Computer System

Subjects: Others
Contributor: HandWiki Xu

Fault-tolerant computer systems are systems designed around the concepts of fault tolerance. In essence, they must be able to continue working to a level of satisfaction in the presence of errors or breakdowns. Fault tolerance is not just a property of individual machines; it may also characterise the rules by which they interact. For example, the Transmission Control Protocol (TCP) is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communications links which are imperfect or overloaded. It does this by requiring the endpoints of the communication to expect packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount. Recovery from errors in fault-tolerant systems can be characterised as either 'roll-forward' or 'roll-back'. When the system detects that it has made an error, roll-forward recovery takes the system state at that time and corrects it, to be able to move forward. Roll-back recovery reverts the system state back to some earlier, correct version, for example using checkpointing, and moves forward from there. Roll-back recovery requires that the operations between the checkpoint and the detected erroneous state can be made idempotent. Some systems make use of both roll-forward and roll-back recovery for different errors or different parts of one error.

Keywords: fault-tolerant systems ; checkpointing ; roll-forward

1. Types of Fault Tolerance

Most fault-tolerant computer systems are designed to handle several possible failures, including hardware-related faults such as hard disk failures, input or output device failures, or other temporary or permanent failures; software bugs and errors; interface errors between the hardware and software, including driver failures; operator errors, such as erroneous keystrokes, bad command sequences or installing unexpected software and physical damage or other flaws introduced to the system from an outside source.[1]

Hardware fault-tolerance is the most common application of these systems, designed to prevent failures due to hardware components. Most basically, this is provided by redundancy, particularly dual modular redundancy. Typically, components have multiple backups and are separated into smaller "segments" that act to contain a fault, and extra redundancy is built into all physical connectors, power supplies, fans, etc.[2] There are special software and instrumentation packages designed to detect failures, such as fault masking, which is a way to ignore faults by seamlessly preparing a backup component to execute something as soon as the instruction is sent, using a sort of voting protocol where if the main and backups don't give the same results, the flawed output is ignored.

Software fault-tolerance is based more around nullifying programming errors using real-time redundancy, or static "emergency" subprograms to fill in for programs that crash. There are many ways to conduct such fault-regulation, depending on the application and the available hardware.[3]

2. History

The first known fault-tolerant computer was SAPO, built in 1951 in Czechoslovakia by Antonín Svoboda.[4]:155 Its basic design was magnetic drums connected via relays, with a voting method of memory error detection (triple modular redundancy). Several other machines were developed along this line, mostly for military use. Eventually, they separated into three distinct categories: machines that would last a long time without any maintenance, such as the ones used on NASA space probes and satellites; computers that were very dependable but required constant monitoring, such as those used to monitor and control nuclear power plants or supercollider experiments; and finally, computers with a high amount of runtime which would be under heavy use, such as many of the supercomputers used by insurance companies for their probability monitoring.

Most of the development in the so-called LLNM (Long Life, No Maintenance) computing was done by NASA during the 1960s,[5] in preparation for Project Apollo and other research aspects. NASA's first machine went into a space observatory, and their second attempt, the JSTAR computer, was used in Voyager. This computer had a backup of memory arrays to use memory recovery methods and thus it was called the JPL Self-Testing-And-Repairing computer. It could detect its own errors and fix them or bring up redundant modules as needed. The computer is still working today.

Hyper-dependable computers were pioneered mostly by aircraft manufacturers,[4]:210 nuclear power companies, and the railroad industry in the USA. These needed computers with massive amounts of uptime that would fail gracefully enough with a fault to allow continued operation, while relying on the fact that the computer output would be constantly monitored by humans to detect faults. Again, IBM developed the first computer of this kind for NASA for guidance of Saturn V rockets, but later on BNSF, Unisys, and General Electric built their own.[4]:223

The 1970 F14 CADC had built-in self-test and redundancy.[6]

In general, the early efforts at fault-tolerant designs were focused mainly on internal diagnosis, where a fault would indicate something was failing and a worker could replace it. SAPO, for instance, had a method by which faulty memory drums would emit a noise before failure.[7] Later efforts showed that, to be fully effective, the system had to be self-repairing and diagnosing – isolating a fault and then implementing a redundant backup while alerting a need for repair. This is known as N-model redundancy, where faults cause automatic fail safes and a warning to the operator, and it is still the most common form of level one fault-tolerant design in use today.

Voting was another initial method, as discussed above, with multiple redundant backups operating constantly and checking each other's results, with the outcome that if, for example, four components reported an answer of 5 and one component reported an answer of 6, the other four would "vote" that the fifth component was faulty and have it taken out of service. This is called M out of N majority voting.

Historically, motion has always been to move further from N-model and more to M out of N due to the fact that the complexity of systems and the difficulty of ensuring the transitive state from fault-negative to fault-positive did not disrupt operations.

Tandem and Stratus were among the first companies specializing in the design of fault-tolerant computer systems for online transaction processing.

3. Fault Tolerance Verification and Validation

The most important requirement of design in a fault tolerant computer system is making sure it actually meets its requirements for reliability. This is done by using various failure models to simulate various failures, and analyzing how well the system reacts. These statistical models are very complex, involving probability curves and specific fault rates, latency curves, error rates, and the like. The most commonly used models are HARP, SAVE, and SHARPE in the USA, and SURF or LASS in Europe.

4. Fault Tolerance Research

Research into the kinds of tolerances needed for critical systems involves a large amount of interdisciplinary work. The more complex the system, the more carefully all possible interactions have to be considered and prepared for. Considering the importance of high-value systems in transport, public utilities and the military, the field of topics that touch on research is very wide: it can include such obvious subjects as software modeling and reliability, or hardware design, to arcane elements such as stochastic models, graph theory, formal or exclusionary logic, parallel processing, remote data transmission, and more.[8]

4.1. Failure-Oblivious Computing

Failure-oblivious computing is a technique that enables computer programs to continue executing despite memory errors. The technique handles attempts to read invalid memory by returning a manufactured value to the program, which in turn, makes use of the manufactured value and ignores the former memory value it tried to access. This is a great contrast to typical memory checkers, which inform the program of the error or abort the program. In failure-oblivious computing, no attempt is made to inform the program that an error occurred.[9] Furthermore, it happens that the execution is modified several times in a row, in order to prevent cascading failures.[10]

The approach has performance costs: because the technique rewrites code to insert dynamic checks for address validity, execution time will increase by 80% to 500%.[11]

4.2. Recovery Shepherding

Recovery shepherding is a lightweight technique to enable software programs to recover from otherwise fatal errors such as null pointer dereference and divide by zero.[12] Comparing to the failure oblivious computing technique, recovery shepherding works on the compiled program binary directly and does not need to recompile to program. It uses the just-in-time binary instrumentation framework Pin. It attaches to the application process when an error occurs, repairs the execution, tracks the repair effects as the execution continues, contains the repair effects within the application process, and detaches from the process after all repair effects are flushed from the process state. It does not interfere with the

normal execution of the program and therefore incurs negligible overhead.[12] For 17 of 18 systematically collected real world null-dereference and divide-by-zero errors, a prototype implementation enables the application to continue to execute to provide acceptable output and service to its users on the error-triggering inputs.[12]

References

1. Fault-tolerant computer system design book contents. Dhiraj K. Pradhan, Pages: 135 – 138 1996 ISBN:0-13-057887-8
2. Formal Techniques in Real-Time and Fault-Tolerant Systems: Second International Symposium, Nijmegen, the Netherlands, January 8–10, 1992, Proceedings By Jan Vytopil Contributor Jan Vytopil, Published by Springer, 1991, ISBN:3-540-55092-5, 978-3-540-55092-1
3. Fault-tolerant computer system design book contents. Dhiraj K. Pradhan, Pages: 221 – 235 1996 ISBN:0-13-057887-8
4. Daniel P. Siewiorek; C. Gordon Bell; Allen Newell (1982). Computer Structures: Principles and Examples. McGraw-Hill. ISBN 0-07-057302-6. http://archive.computerhistory.org/resources/text/bell_gordon/bell.computer_structures_principles_and_examples.1982.102630397.pdf.
5. "The STAR (Self-Testing And Repairing) Computer: An Investigation Of the Theory and Practice Of Fault-tolerant Computer Design". https://www.computer.org/csdl/proceedings/ftcsh/1995/7150/00/00532604.pdf.
6. Ray Holt. "The F14A Central Air Data Computer, and the LSI Technology State-of-the-Art in 1968". http://www.firstmicroprocessor.com/documents/lsistate-97.pdf
7. Fault tolerant computing in computer design Neilforoshan, M.R Journal of Computing Sciences in Colleges archive Volume 18 , Issue 4 (April 2003) Pages: 213 – 220, ISSN 1937-4771
8. Reliability Evaluation of Some Fault-Tolerant Computer Architectures By Shunji Osaki, Toshihiko Nishio Published by Springer, 1980 ISBN:3-540-10274-4, 978-3-540-10274-8
9. Rinard, Martin; Cadar, Cristian; Dumitran, Daniel; Roy, Daniel M.; Leu, Tudor; Beebee, William S. (2004), "Enhancing server availability and security through failure-oblivious computing", Enhancing server availability and security through failure-oblivious computing, Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation, 6, Berkeley, CA: USENIX Association
10. Durieux, Thomas; Hamadi, Youssef; Yu, Zhongxing; Baudry, Benoit; Monperrus, Martin (2018). Exhaustive Exploration of the Failure-Oblivious Computing Search Space. doi:10.1109/ICST.2018.00023. https://arxiv.org/pdf/1710.09722.
11. Keromytis, Angelos D. (2007), "Characterizing Software Self-Healing Systems", in Gorodetski, Vladimir I.; Kotenko, Igor; Skormin, Victor A., Characterizing Software Self-Healing Systems, Computer network security: Fourth International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, Springer, ISBN 3-540-73985-8, https://books.google.com/books?id=N2uIjckxHSoC&pg=PA27&dq=failure-oblivious&hl=en&ei=gmlSTc7aD4nUvQPZwMXPCQ&sa=X&oi=book_result&ct=result&resnum=2&ved=0CCkQ6AEwAQ#v=onepage&q=f oblivious&f=false
12. Long, Fan; Sidiroglou-Douskos, Stelios; Rinard, Martin (2014). "Automatic Runtime Error Repair and Containment via Recovery Shepherding". Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14'. New York, NY, USA: ACM. pp. 227–238. doi:10.1145/2594291.2594337. ISBN 978-1-4503-2784-8. https://dx.doi.org/10.1145%2F2594291.2594337