System Availability Benchmarking - A Survey

Jeremy Langston
School of Electrical and Computer Engineering
Tennessee Technological University
Cookeville, Tennessee 38505
Email: jwlangston21@tntech.edu

***Abstract*—This survey report highlights the main topics associated with availability benchmarking of computing and data storage systems. Compared to performance benchmarking, availability is still in its infancy. A widely adopted benchmarking framework has yet to be designed. This is partly due to the complexity of such a task and the problems associated with it. Numerous factors have to be taken into account, and there is a debate as to how to include these factors in the benchmark. This paper will give the reader a basic understanding of the concepts associated with availability, some of the current availability representations, and briefly discuss some current research being done in this area.**

I. Introduction

The availability of a system can be defined as the system's continued ability to provide application level services to its users [1]. That is, availability is focused at the usability level - can the system provide its services at all times to the users. A further analysis into this concept states that high-availability is having access to data and applications whenever needed with an acceptable level of performance [2].

Reliability and serviceability often go hand-in-hand with availability, and sometimes the differences are not apparent. [1] gives a good explanation of reliability versus availability. Reliability provides a metric of how often a component fails, while availability includes the effects of downtime any failures produce. Serviceability is synonymous with recoverability - the characteristic of a system's reaction to faults, errors, and maintenance.

An availability benchmark is a test, or series of tests, that is run on a system or a simulated system in order to be compared against other systems, typically on the basis of uptime, or downtime [4]. The benchmark can also test a system's responsiveness of, and recovery to, a variety of different faults. Currently the majority of availability benchmarking is done by either a vendor or manufacturer. These benchmarks are often proprietary and subjective, not standardized and objective, which cause them to lose credibility during comparisons. Users and developers need a way to compare a set of systems using the same testing conditions. Even more importantly, these benchmarks must exude uniformity and repeatability for them to be accepted [5], as well as comprehensiveness, representativeness, fairness, relevance, and independence [6].

Historically, the world of benchmarking has been dominated by performance testing. Not until recently has system availability really gained recognition and respect. As noted in [3], system downtime can exceed $500,000 per hour, stressing the importance of a highly available infrastructure. Each type of system has different availability requirements, contingent on their usages. For example, on-board avionic computer systems have entirely different standards than an engineering simulation system, or even a network storage system. The requirements are drawn up from users, customers, and sometimes developers. To fulfill the requirements, a benchmark could define a system based on its availability, from which a system could be selected. In another application, a benchmark could determine what effect a change in the system would manifest.

There are many ways to classify a system, which can change the benchmark dramatically. Systems can be categorized by how they are arranged, what their functions are, how they perform their functions, and so on. The high performance commercial benchmarking community is familiar with the idea of transactional systems and backup systems. Transactional systems seen in banks hold highly the amount of traffic they can sustain while keeping the system secure and available. Backup systems will focus on redundancy; keeping data stored and up to date is of the highest importance. It is easy to see how the implementation domains are different and as such should be taken into account. Storage systems can also be vastly different in their implementations. Take for example a traditional RAID (redundant array of independent disks) setup on a commodity system. The characteristics of that system would be quite distinguishable from that of a dedicated NAS (network addressed storage) system. Further classifications can be made by the level of availability required. For example, an orbiting satellite's computer system will have different requirements than that of an email system. An email server may allow for better performance at the sacrifice of availability due to a low reaction time of operators. However, sending an operator to fix the system on a satellite is very costly, therefore uptime is of utmost importance.

The remainder of the paper is organized as follows: Section 2 lists a few of the metrics commonly associated with availability. Section 3 details some of the factors pertinent to the topic such as typical hardware failures. Section 4 briefly discusses different modeling techniques for testing. Section 5 explains issues related to workloads in fault-injection testing. Section 6 covers a few of the known problems that present roadblocks and misrepresentation. Lastly, section 7 presents what other research is being done and section 8 concludes the paper.

II. Metrics

Availability metrics can be a source of confusion, misrepresentation, and disagreement. Some have taken a metric such as MTBF and misconstrued it to mean something quite different. It is important to have a standardized metric or metrics that are well understood and accepted. Below are a few of the most widely used representations.

A. Mean Time Between Failures - MTBF

A reliability measure, this calculation is the average time period between two consecutive component failures that require repair [1]. The higher the MTBF, the higher the availability. Since the MTBF is an average of times, it cannot be used to give a time of failure for a component. Consider a hard drive with a MTBF of 50,000 hours. This means, *on average*, the hard drive will work for 50,000 hours before a failure. This does not imply that the time period is static. The hard drive could fail 5 hours after being put online, but the next failure may occur 99,995 hours later, giving an MTBF of 50,000 hours. The MTBF can be viewed on the hardware, software, and system levels. [1] points out a misconception that introducing redundancy will lower the MTBF. This may be true for system MTBF overall, but the opposite is true for hardware or software MTBF as there are more components in which a failure may occur.

B. Mean Time Between Interruptions - MTBI

MTBI is very similar to MTBF. One difference is between an interruption and a failure. An interruption is temporary and does not require repair. Another difference is that MTBI is directly related to the end user, making it a closer representation to availability than reliability.

C. Mean Time To Repair - MTTR

This metric depicts the duration of the repair and recovery time. This means that if the time between the initial failure and subsequent repair putting the system back to working order is 10 minutes, then the MTTR is 10 minutes. Repair can come in the form of an automatic recovery (e.g. RAID rebuilding) or manually by an operator.

D. Uptime

The most cited metric is uptime, often expressed as a percentage ratio. This is a high level view of the system, as it pertains to availability to the end user or application level. Uptime ratio is calculated as follows:

$$UptimeRatio = \frac{MTBF}{MTBF + MTTR} * 100\% \qquad (1)$$

The availability of a system is often referred to by the number of nines in the uptime calculation result. Currently, systems are seeing as high as Five 9's, or 99.999% uptime, or even higher. By calculating the downtime ratio and multiplying by some time period, one can calculate the amount of time the system is offline for that time period. Downtime ratio is merely $(1 - UptimeRatio)$. For example, a system with Five 9's availability has a downtime of 5.256 minutes per year.

Hardware	Component failure, power/cooling/environment, arrangement.
Software	Data corruption, driver timeout, incorrect specifications.
Process	Human element, maintenance, security attacks, upgrades.

TABLE I
EXAMPLE FACTORS AFFECTING SYSTEM AVAILABILITY.

III. Availability Factors

There are numerous factors that either directly or indirectly affect system availability. Familiarity with these factors and their faults will lead to a more robust benchmark framework. A common categorization is labeling the factors as being hardware, software, or process related [7]. Table 1 gives a sample listing of these factors. There are other classifications that can be performed, such as frequency, severity, redundancy, or fault location.

Typically, hardware faults are the least frequent, followed by software faults, and then process engagements. This is not necessarily the same for every system. The more hardware components that are active in a given system, the higher the frequency of failure. To see this, consider a RAID array. Let f be the frequency of a single disk fault and n be the number of disks in the array. For a system running on a single disk, the frequency of disk faults would be 1(f). For a system running, say, 3 disks, the frequency of hard disk faults clearly triples to 3(f). This is a very simplistic example, but illustrates the point. Process faults represent a reportedly high amount of overall faults mainly due to one issue: human interaction [6].

The human element is the most difficult factor to include in a benchmark. Here, realism is vital. Not only do the faults requiring human intervention need to be realistic, the way in which humans interact need to be realistic. In a leading paper discussing the human factor, [8] states the difficulties involving modeling human operators. Two key problems were determined: human variability and repeatability. All operators have differing skill sets and knowledge. In order to properly test the frequency of human error, the operators must all be of equivalent abilities. Clearly an expert in the field will make fewer mistakes than a technician. The authors chose to use experts to perform the benchmarking to exclude any unnecessary faults. However as long as the same level of operators do the testing on all systems to be benchmarked, there will be no real problem. The authors stated the problem with repeatability was that once testers were exposed to a fault, they learned how to solve it. The next time the test was run, they would already know what to do. They referred to this as the *learning curve*. Tasks and problems the operators would face are initial configuration, reconfiguration, monitoring, diagnosis and repair, an preventative maintenance.

IV. Modeling

A theoretical approach to availability benchmarking is through system modeling. This can take the form of mathematical formulae, state diagrams, and simulations. State diagrams graphically show the different states of a computer system and

the states that they can transition to. This analysis is easy to comprehend, but suffers from exponential expansion due to the number of components that should be included. The Markov Chain has similarities to state diagrams, but expands them giving weights, or probabilities, to each transition. However, they are far more complex and, thus, prone to mistakes.

Simulation programs for modeling systems have been written by universities and companies. One such company, Isograph, has an array of programs that will analyze system faults, reliability, costs, and various other areas, giving comprehensive, detailed reports [9].

V. Faultloads

One of the most common ways to benchmark a system is with a workload. Performance benchmarks have CPU, memory, I/O, and other intensive programs for testing different areas of a system. These intensive programs use workloads specialized for testing responses to integer and floating point arithmetic, network throughput, etc. Availability benchmarks may work in the same way. Here the workload contains faults that will characterize real-world faults. A faultload can contain a corrupted system call or an invalid repair procedure or even disrupting the power feeding the system. The faultload can be used on simulated systems, physical hardware, or software [10]. As stated previously, there are fewer hardware faults and failures than that of software. It turns out there are far fewer which leads to reducing the frequency of hardware faults in the faultload.

There are two common ways of obtaining faultloads: system traces and hypothetical failures. System traces are logs that show the daily functions and anomalies of a system. They can be retrieved from IT centers with systems already in place. However, for a system currently in development or soon after, these logs have not been written yet. Also, these logs do not necessarily depict the way a different system may behave. This poses a problem: how can a faultload be designed without previously seeing how the system will behave? Typically a company will make a particular model of a system and make advances to it without starting from scratch. When this is the case, system logs can be used for faultloads and be still quite accurate. Hypothetical faultloads can stray from the system's real-world failure characteristics. The faults used will often be generic, but can go in great detail if the faultload designer knows the inner workings of the system. One gray area to faultload representativeness is the level of independency from the system. For a benchmark to be used universally, the faultload cannot be too overly detailed.

VI. Problems

In order for the faultload to be effective, it must represent real-world scenarios. [12] addresses the debate on fault-injection representativeness. The topic is whether a system should be measured based on responses to arbitrary faults. To understand what an arbitrary fault may be, consider how a typical memory system is built. A memory module often used in server applications uses parity bits to determine corruption. Upon retrieval of the memory location that is corrupted, the system is notified and the necessary changes are made (often reloading from disk). This is built in the hardware level, and the software need not know how to check for memory corruption. If the faultload designer is not careful, he or she may inject the fault at a level where a check would have already been made, such as after checking the parity bits but before retrieving and using the memory. Software could have caught the corrupted memory, but often designers forgo the extra check for speed and efficiency. In most practical instances, the software can assume the hardware is faultless. The same goes for a non-atomic instruction block (meaning it can be interrupted). A check is done on some data and then the data is used. Injecting a fault at the instant the check is completed and good, but before the data is processed, can be considered arbitrary.

Fault-injection benchmarks are used to test a systems reaction to faults. As [4] points out, fault-injection techniques leave out two vital parts of an overall benchmark. These benchmarks typically inject far more faults than would occur naturally. The rate at which faults occur is a difficult number to assess, but may be sampled from traces. Secondly, these benchmarks do not typically test recovery. This issue has been addressed in [11].

Another problem is comparing two architecturally different systems [7]. This is especially problematic benchmarking across different system designing companies. This issue is referred to as the benchmark's portability. For fault injection, [19] addresses this problem stating that the system-specific and non-specific methods should be separated. The non-system-specific method could be done using software implemented fault injection at a high level. Tsai et. al. approach the problem by implementing a two stage fault injection process: a high-level portion and a low-level portion. The high-level part is intended to be highly portable across UNIX systems by selecting fault parameters to be used by the low-level part. The low-level part is implemented as a device driver in order to beat fault-tolerant systems.

VII. Related Work

Below are some of the previous tests and benchmarks, as well as ones currently in research.

A. Physical Tests

Mendosus is an emulation test-bed for fault-injection on SANs applications that was produced at Rutgers University [14]. A SAN is emulated to the PCs connected to it. The application on each PC is tested through faults injected by the emulated SAN.

Messaline [15] is a fault-injection system that uses physical contact perturbations to induce failures. Such injections are done at the pin level with active probes and socket insertion [13]. Active probes are devices that make contact with the physical hardware and alter the electrical characteristics. Socket insertion is putting a piece of hardware that lies between the PCB and a chip. This socket can then cause

failures by manually changing the data on the chip (e.g. changing a 1 to a 0).

MARS, Maintainable Real-time System, is a fault-tolerant architecture built to stand up against rigorous testing. Three physical tests are performed by [16]: heavy-ion radiation, pin-level faults, and electromagnetic interference (EMI).

B. Software Tests

FERRARI, Fault and Error Automatic Real-time Injection, works on the software level by using software traps [17].

Ftape, Fault Tolerance and Performance Evaluator, is a tool with a controllable workload generator that allows faults to be injected at high levels of usage on fault-tolerant systems [18].

Crashme is a test with one main objective: to crash the operating system. The program passes illegal instructions, data, and operations to the operating system to see if they are caught [19].

Define, Doctor, and Fiat are three other tools that can be used in software fault injection testing [19].

C. Hybrid Tests

R-Cubed [4], R^3, is a benchmark framework designed at Sun Microsystems, Inc. They propose a hierarchical approach that addresses three key topics: rate (frequency of faults and maintenance), robustness (ability to detect and handle events), and recovery (speed system returns to operational state). Each of these three topics has its own metric, and that metric can be decomposed into either fault or maintenance event classifications.

As discussed in the Problems section, [18] implements a benchmarking framework for three Tandem TMR-based fault-tolerant prototype systems. The authors stress-tested the systems, introducing faults along the way using the FTAPE injection tool. Three different workloads (CPU, memory, and I/O intensive) were run at varying frequencies. They collected data such as fault-to-error ratios, performance degradation, and catastrophic incidents.

VIII. Conclusion

High availability is a requisite for today's high-end systems where downtime can result in catastrophic losses. This report has stressed the need for an accepted computing and data storage system availability benchmark frameworks. Benchmarking results will allow for an indiscriminant comparative analysis between systems. To this end, companies will be better suited for choosing the correct system to fit their needs and developers and researchers will have a metric to determine the best course of action for system design.

References

[1] E. Vargas, "High Availability Fundamentals", Sun BluePrints Online, Enterprise Engineering, Sun Microsystems, Inc., 2000.

[2] D. Brock, "A Recommendation for High-Availability Options in TPC Benchmarks", Data General.

[3] Sombers Associates, Inc., and W. H. Highleyman, "Let's Get an Availability Benchmark", The Availability Digest, 2007.

[4] J. Zhu, J. Mauro, and I. Pramanick, "R-Cubed (R^3): Rate, Robustness, and Recovery - An Availability Benchmark Framework", Sun Microsystems, Inc., 2002.

[5] K. Kanoun, H. Madeira, and J. Arlat, "A Framework for Dependability Benchmarking", *DSN Workshop on Dependability Benchmarking*, LAAS-CNRS, Toulouse, France and DEI-FCTUC, University of Coimbra, Coimbra, Portugal, 2002.

[6] D. Oppenheimer, A. Brown, J. Traupman, P. Broadwell, and D. Patterson, "Practical Issues in Dependability Benchmarking", University of California at Berkeley.

[7] D. Wilson, B. Murphy, L. Spainhower, "Progress on Defining Standardized Classes for Comparing the Dependability of Computer Systems", *DSN Workshop on Dependability Benchmarking*, 2002.

[8] A. Brown, C. Chung, D. Patterson, "Including the Human Factor in Dependability Benchmarks", University of California at Berkeley, 2002.

[9] Isograph, Inc., http://www.isograph-software.com.

[10] J. Arlat, Y. Crouzet, "Faultload Representativeness for Dependability Benchmarking", *DSN Workshop on Dependability Benchmarking*, LAAS-CNRS, Toulouse, France, 2002.

[11] J. Zhu, J. Mauro, I. Pramanick, "System Recovery Benchmarking", *DSN Workshop on Dependability Benchmarking*, Sun Microsystems, Inc., 2002.

[12] P. Koopman, "What's Wrong with Fault Injection as a Benchmarking Tool?", *DSN Workshop on Dependability Benchmarking*, ECE Department & ICES, Carnegie Mellon University, 2002.

[13] M. Hsueh, T. Tsai, R. Iyer, "Fault Injection Techniques and Tools", *IEEE Computer*, Volume 30, Number 4, Pages 75-82, 1997.

[14] X. Li, R. Martin, K. Nagaraja, T. Nguyen, B. Zhang, "Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services", In 1st Workshop on Novel Uses of System Area Networks (SAN-1), Rutgers University, 2002.

[15] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. Fabre, E. Martins, D. Powell, "Fault Injection for Dependability Validation: A Methodology and Some Applications", In IEEE Transactions on Software Engineering, Volume 16, Issue 2, Pages 166-182, 1990.

[16] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, G. Leber, "Evaluation of the MARS Architecture by means of Three Physical Fault Injection Techniques", LAAS-CNRS, Chalmers University of Technology, Technical University of Vienna, 1995.

[17] G. Kanawati, N. Kanawati, J. Abraham, "FERRARI: A Flexible Software-Based Fault and Error Injection System", In IEEE Transactions on Computers, Volume 44, Issue 2, 1995.

[18] Tsai, T., R. Iyer, "Measuring Fault Tolerance with the FTAPE Fault Injection Tool", In Proceedings Eighth International Conference on Modeling Techniques and Tools for Computer Performance Evaluation, Heidelberg, Germany, 1995.

[19] T. Tsai, R. Iyer, D. Jewett, "An Approach towards Benchmarking of Fault-Tolerant Commercial Systems", University of Illinois, 1996.