The Basics of Dependability
CS444a

George Candea
Stanford University
candea@cs.stanford.edu

1. What is Dependability?

We have a general notion of what it means to be dependable: turn your homework in on time, if you say you'll do something then do it, don't endanger other people, etc. Machines however do not understand vague concepts—we need a crisper definition when it comes to computer systems. Laprie [1991] refers to dependability as the trustworthiness of a computer system, and defines it in terms of 4 attributes:

- **Reliability** is the aspect of dependability refering to the *continuity* of a system's correct service: if my desktop computer offered service without interruption (i.e., without failure) for a long time, then I would say it was very reliable.
- **Availability** refers to a system's *readiness for usage*: if the Google search engine answers my queries every time I submit a request, then I can say it is highly available. Note that availability says nothing about the system when you are not asking it for service: if `google.com` was unreachable whenever nobody accessed the search engine, it would make no difference with regards to availability, but it would certainly impact the service's reliability.
- **Safety** has to do with systems *avoiding catastrophic consequences* on their environment and/or operators. The control system for a nuclear power plant is safe if it avoids nuclear accidents; an airplane computer system is safe if it avoids plane crashes. As businesses rely increasingly more on computing infrastructures, the notion of safety can be extended to domains that do not directly impact life. For example, a large scale failure can drive a company out of business, and that can be considered an issue of "business safety." Hence, a catastrophe in the context of computer safety is generally an instance where the consequence is considerably worse than the benefit we obtain from the system being up.
- **Security** captures the ability of a system to *prevent unauthorized access* (read and/or write) to information. The E*Trade online brokerage system is secure if I have no way to see and/or modify account information I am not allowed to. Although many perceive denial-of-service as a security problem, its first-order effect is actually an attack on availability and reliability, in the context of this definition.

Thus, if a computer system is reliable, available, safe, and secure, then we say it is dependable.

Computer people like to pretend our field is a science, so there is a lot of mathematics that goes hand in hand with these attributes. For now, all you need to know is that they can be defined in terms of random variables, and we can apply interesting probability techniques to them.

You can think of reliability as a random variable $Rel(t)$ representing the probability that a system does not fail in the time interval $[0, t)$. If you compute the expected value of $Rel(t)$, you get what is known as mean-time-to-failure, or MTTF.

In order to define availability mathematically, we use the notion of recoverability, a metric that captures how quickly a system can be recovered once it has failed: $Rec(t)$ is the probability that the system is back to normal t time after failure. The expected value of $Rec(t)$ is the mean-time-to-recover, or MTTR.

Instantaneous availability can then be thought of as $A(t)$, the probability that a system delivers correct service at time t, where "correct" is defined by a specification or a service-level agreement (SLA). The expected value of $A(t)$ is what we often refer to as availability; this can be computed in terms of the expected values of $Rel(t)$ and $Rec(t)$:

$$\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \tag{1}$$

You will sometimes see mean-time-between-failures (MTBF) used instead of MTTF. Instead of capturing the mean time to a new failure *after* the previous failure has been repaired, MTBF captures the mean time *inbetween* successive failures. Therefore, MTBF = MTTR + MTTF, and availability then becomes (MTBF-MTTR)/MTBF.

In the computer industry, an often used buzzword is the notion of "nines of availability." This corresponds to the total number of nines in the availability number given as a percentage, i.e. 99.99% availability corresponds to "4 nines," 99.999% is "5 nines," etc. If you do the math, you can come up with a pretty simple rule of thumb: 5 nines means a maximum of ≈5 minutes of downtime per year, 4 nines ≈50 minutes, 3 nines ≈500 minutes, 2 nines ≈5,000 minutes, and so on. However, when a company claims that its device or software is five-nines reliable, it is talking about a complicated mathematical calculation that often has received a lot of "help" from the marketing department. The formula used by one vendor is very rarely the same with the formula used by other vendors. So, when someone tells you they can offer five nines of availability, be very skeptical—the claim is usually devoid of any practical meaning.

To get a sense of what these nines mean, consider the following examples: a well-run Internet service, like Amazon.com, offers around 2 nines of availability; a typical desktop or server will likely be around 3-nines available, an enterprise server will offer 4-nines availability, and carrier-grade phone lines will usually be 5-nines available. The goal of carrier phone switches is around 6-nines. There are many problems with expressing availability as a percentage, including the period over which availability is measured, what is considered up and what is down, etc. See Fox and Patterson [2002] for a discussion of these problems.

2. Historical Evolution

The different aspects of dependability have varied in importance over the course of time, depending on then-contemporary technological problems. The first general purpose computer, ENIAC, became operational during World War II and performed 18,000 multiplications per minute. The ENIAC had almost 18,000 vacuum tubes, resulting in an overall MTBF of 7 minutes. Since the MTBF was the same order of magnitude as a reasonably complex computation, reliability of this system was clearly the foremost problem. The key to keeping it running was to reduce repair time (MTTR): a staff of 5 people were patrolling the innards of the 30-ton monster on roller skates and replacing tubes as soon as they blew out. Recently, Van der Spiegel et al. [2000] reproduced the original ENIAC on a single CMOS chip, 7.44mm x 5.29mm (i.e., one tenth the size of a postage stamp), containing about 180,000 transistors (ten times the number of vacuum tubes in the original ENIAC). The reliability of the ENIAC-on-a-chip is many orders of magnitude higher than the original. Refer to Siewiorek and Swarz [1998] for a thorough treatment of reliability in computer systems.

Hardware reliability improved significantly over the two decades following ENIAC. In the early sixties, most computers ran batches of jobs submitted by users to computer operators, who scheduled the jobs to keep the computer fully occupied. Turn-around time for each job was on the order of hours. Hardware failures would cause batch jobs to fail, thus having to be re-run; brief periods of unavailability, however, were not a big deal. With the advent of on-line, interactive computers (the TX-2 developed at MIT's Lincoln Laboratory was one of the first), availability suddenly became important: users walking up to a terminal expected the system to be available to complete their requests. This quest for high availability continues today, as the Web has taken interactivity requirements to a new level, where services need to serve thousands of customers every second. Marcus and Stern [2003] is an excellent reference for how such systems are designed today.

As MTTF and MTTR of hardware and software improved, interactive systems were becoming reasonably available, and computers started being used for applications where traditionally humans or mechanical devices had been doing the job. In 1972, NASA tested a modified Navy F-8 Crusader with a digital fly-by-wire[1] system, which was replacing equivalent analog systems. About a decade later, the MD-11 was the first commercial aircraft to adopt computer-assisted flight controls, followed by the Airbus A320. With the advent of applications involving human lives, safety became an important issue. Writing safe software is an ongoing challenge; a case in point is the US Patriot missile defense system, designed to intercept and destroy incoming missiles in mid-air prior to reaching their target. As reported by the G.A.O. [1992], the Patriots used during the Gulf War had a bug: if they ran for more than eight hours without being rebooted, they miscalculated trajectories of incoming missiles. On Feb. 25, 1991, this bug caused a Patriot to miss an incoming Scud missile, which slammed into US military barracks in Saudi Arabia, killing 28 soldiers and wounding 98. See Leveson [1995] for detailed analyses of software systems and their role in the safety of larger industrial and military systems.

The nineties saw a tremendous growth in distributed systems and the Internet, largely driven by the Web. As various organizations flocked to the Internet, putting their computer systems online, so did the community of crackers. A large increase in break-ins into computer systems and networks, accompanied by information theft, has made the topic of computer security a high priority topic, and continues to be so today. Kevin Mitnick was one of the first cases to hit the media spot light: arrested by the FBI in 1995, Mitnick was charged with stealing $1 million worth of sensitive project data from computer systems, snagging thousands of credit card numbers from online databases, breaking into the California motor vehicles database, and remotely controlling New York and California's telephone switching hubs on various occasions. To learn more about security, read Anderson [2001], a great treatise on the design of secure computer systems.

3. Classifications

In the world of Internet systems, it is now widely accepted that service provision and receipt should be governed by an agreement. Such contracts, called service level agreements (SLAs), define the parameters of the service, for the benefit of both the provider and the recipient. For example, at the beginning of 2003, MCI's SLA for DSL broadband service guaranteed a minimum of 99% monthly availability (i.e., no more than 7.5 hours of downtime in any given 31-day period) based on trouble ticket time. Downtime begins when a trouble ticket is opened with MCI support for an outage, and ends when the service is back up and the trouble ticket is closed. Should MCI not conform, the customer is entitled to receive a credit to her account for one day's worth of the contracted monthly recurring charge.

[1] In traditional aircraft, pilots controlled flight surfaces (rudder, flaps, etc.) directly through pulleys and hydraulic servo systems. In fly-by-wire planes, the pilot issues commands to a computer, which then decides what and how to actuate. In some cases, the fly-by-wire system may decide to override the pilot, if the command is deemed unsafe by the computer. This introduces a highly non-linear relationship between pilot controls inputs and the movement of the controlled subsystems.

Creation of an appropriately focused and enforceable SLA is a daunting task, but many view it as a necessary evil. Part of the problem is the lack of a widely accepted and understood vocabulary, as well as the difficulty of expressing the relevant dependability events and concepts in terms of system details.

3.1. Faults, Errors, and Failures

The notions of fault, failure, and error are generally used rather loosely, yet they can be defined quite precisely. A **failure** is the deviation of a system from its specification or SLA. If we consider the specification of a network system to require the timely delivery of network packets, then any dropped packet would be considered a failure. In the MCI SLA example, if the network is down for more than 7.5 hours in any 31-day period, it is deemed to have failed, yet if it had been down for only 7 hours, it would not have officially failed. This distinction suggests some of the major simplifications that need to be made in order for SLAs to even exist.

An **error** is the part of the system's state that leads or may lead to an observed failure; for an Internet service provider like MCI, corrupt routing tables could cause downtime that results in packet delivery failure. Finally, a **fault** is the deemed cause of an error, that lead to a failure. For a network outage, this fault could be an optic fiber severed by a backhoe (environmental fault), a fried motherboard on a central router (hardware fault), a bug in the routing software (software fault), or a human operator that misconfigured routing tables (human fault). An example of a human fault occured on January 23, 2001, when a technician misconfigured routers on the edge of Microsoft's DNS network. This resulted in many of Microsoft's sites, including the MSN ISP business, to become unreachable. 22.5 hours later, the changes were undone and the network recovered.

The distinction between fault, error, and failure explains how "fault tolerance" is different from "error correction," and how "fault injection" is different from "error injection." It is important to note that the failure of one system often becomes a fault for another system. If a spilled can of soda (fault) causes a disk drive controller to burn and stop responding to SCSI requests (disk failure), this could be perceived as a fault by the database program reading from the disk. The database program may end up corrupting data on another disk it is accessing (data error), and ultimately supply incorrect data to the user or simply crash, both of which constitute database failures.

3.2. Types of Faults and Failures

Software bugs are faults; they are unavoidable in software systems that are complex and often beyond human understanding. Quantum physicists, in their quest for finding the simple explanation to the Universe, have uncovered a whole zoo of particles; perhaps that is why, in hacker jargon, we name bugs after famous quantum physicists and mathematicians. The most famous type of bug is the **Heisenbug**—a bug that stops manifesting, or manifests differently, whenever one attempts to probe or isolate it, particularly when doing so with a debugger. Bugs due to corrupt memory arenas or corrupt stacks often behave this way. The person who coined this term was Bruce Lindsay, who one dark, smokey night during the days of CAL TSS at UC Berkeley (1960's) was struck by the similarities between Heisenberg's Uncertainty Principle and the bug behavior he was witnessing: the closer Bruce was getting to determining the location of the bug, the more erratic the bug was behaving.

The exact opposite of a Heisenbug is a **Bohrbug**, named this way due to its similarity to the easy-to-understand Bohr atom model taught in high schools. A Bohrbug is reproducible and, therefore, relatively easy to find, understand, and fix.

Sometimes, a bug does not manifest until someone reading the source code or using the program in an unusual way realizes that the program should never have worked, at which point the program promptly stops working for everybody until fixed. Such a bug is called a **Schrödingbug**, after Schrödinger's famous Cat thought-experiment,

in which the subject is neither dead nor alive until someone actually notices one state or the other. The phenomenon is not entirely logical, but it is known to occur; some programs have harbored latent Schrödingbugs for years. One might even be tempted to catalogue the recent flurry of Microsoft Windows security bugs as Schrödingbugs, because users are blissfully unaware of the problem until someone discovers the problem, blows the horn, and eager crackers swarm to exploit the vulnerability. Whether these are indeed Schrödingbugs is left up to the reader to decide.

Finally, a bug whose underlying cause is so complex and obscure, that it appears to be entirely nondeterministic, is called a **Mandelbug**. This definition is in apparent analogy to the Mandelbrot set in mathematics: a fractal defined as the set of points in the complex number plane that obey a certain property. The well-publicized Pentium floating point bug of a few years ago, which caused certain divisions to be erroneous, is an example of a fault that could lead to Mandelbugs in scientific software, causing it to malfunction in apparently haphazard ways. This type of bugs are becoming increasingly frequent, as the number of lines of code in software stacks grows at a staggering rate.

Failures are often categorized along two different axes: one with respect to their behavior in time, and another one with respect to their global effects. For the "behavior in time" axis, consider the accidental cutting of an Ethernet cable: this generally results in a **permanent** failure of the subnet that depends on that cable, because the network will not come back up before the cable is replaced. A failure that appears on occasion and seems non-deterministic, in that it is not easily reproducible, is called **intermittent**. Such intermittent behavior may result, for instance, from a dependence on workload or other activity: the Patriot example discussed in section 2 is an intermittent failure, because it is only apparent when a missile comes in and the system has been running for longer than 8 hours. Finally, some failures are **transient**, i.e., once they manifest, retrying or waiting for a while can generally resolve the issue. Wide area network failures are often perceived by end users as transient failures, because the algorithms running in the routers can find alternate routes for the packets, that circumvent the source of the failure.

With respect to the behavior of a failed system, we often refer to systems as fail-stop, fail-fast, or Byzantine. A **fail-stop** system is one that, once failed, does not output any data[2]; this is by far the "best" kind of system, because it is easy to handle fail-stop behavior.

By contrast, **Byzantine** systems do not stop once they fail, and instead return wrong information. The term was first used by Lamport et al. [1982] when describing a distributed system with faulty nodes that send false messages. Apparently generals and officials in the Byzantine Empire were so corrupt, one could never count on what they said. Byzantine failures are the most difficult category of failures to deal with, and unfortunately they are quite frequent. Any time corrupt data is sent to a client, it is a Byzantine failure; if a program confirmed it committed a certain datum to storage, but due to a race condition has failed to do so, it exhibits a Byzantine failure. To make an analogy to every day life, think about dealing with liars versus dealing with people who can only say the truth or shut up. Lynch [1996] provides a voluminous set of elegant algorithms to deal with Byzantine nodes in distributed systems; Castro and Liskov [1999] discuss an implementation of an NFS service that tolerates Byzantine failures.

A **fail-fast** system is one that achieves fail-stop behavior very soon after failing, i.e., it behaves in a Byzantine way for only a short amount of time.

[2]Some authors call this mode fail-silent, and use the term of fail-stop to denote services whose activity is no longer perceptible to users and deliver a constant value output.

3.3. Detection, Diagnosis, Isolation, Repair, Recovery

In order to recover from a problem, one must first know a problem exists. Then, one must figure out what the problem is, prevent it from spreading further, and fix it. The way systems typically deal with faults is in accordance to this general model.

Fault detection is the process of discovering the presence of a fault, usually by detecting the error that fault created. For example, using various parity and/or Hamming codes enables computers to detect when data has been corrupted, whether on disk, in transit on a network, or in a bad memory chip (this is what ECC RAM does). **Fault diagnosis** is the next step, by which the detected error or fault, together with other information, may be used to locate the fault. In triple-modular redundant (TMR) systems, a voter compares the output of three components performing the same computation; in the case of disagreement, the voter declares the minority to have failed and returns the majority result. TMR systems are therefore fail-stop[3] and provide straightforward detection and diagnosis, that allows operators or software to fix the faulty modules.

Once a fault has been detected and diagnosed, the desire is to isolate that fault and prevent it from spreading to otherwise-healthy parts of the system. This is called **fault isolation**, and it is often achieved by using fail-stop components: when a disk fails, rather than returning erroneous data, it returns an error that prevents the read request from completing; this prevents the spread of corrupt data. An alternative way is to put up solid "walls" between parts of the system, to prevent potential faults from propagating: isolate programs onto separate hardware, run them inside virtual machines, etc. Truly fail-stop behavior is difficult to implement, so engineers generally aim for fast detection, diagnosis, and isolation, which leads to fail-fast behavior.

Repair is the process of eliminating the cause of observed failure, and this can take various forms: replacement, reconfiguration, etc. If a disk fails in a redundant array of inexpensive disks (RAID), the operator can open the storage closet, pull out the bad disk, and replace it with a new one; this repairs the RAID.

Following repair, the sytem performs **recovery**—the process of bringing the system back to normal operation. Recovery can take the form of reinitialization, reintegration, etc. In the example above, the RAID will regenerate the data that was on the faulty disk and write it to the new disk; once it completes, the RAID is said to have recovered.

These steps of detection, diagnosis, isolation, repair, and recovery are not always required. For instance, once a fault has been diagnosed, fast recovery will effectively contain the fault, because it doesn't have time to propagate. It is also possible to go from detection straight to recovery: if your desktop computer is frozen (thus, you detect that something went wrong), you reboot the machine without concerning yourself with diagnosing the fault and isolating it. The observation that you can often recover without knowing what went wrong is a fundamental tenet of my research group's work; we will survey some of these ideas in future lectures.

Bibliography

Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, 2001.

Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, New Orleans, LA, February 1999.

Armando Fox and David Patterson. When does fast recovery trump high reliability? In *Proc. 2nd Workshop on Evaluating and Architecting System Dependability*, San Jose, CA, 2002.

[3]The astute reader will have noticed that this is only true in the case of a single component failure. If two or more components failed in the same way or colluded, they could fool the voter and the TMR system would exhibit Byzantine behavior.

G.A.O. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. Technical Report of the U.S. General Accounting Office, GAO/IMTEC-92-26, GAO, 1992.

Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

Jean-Claude Laprie, editor. *Dependability: Basic Concepts and Terminology*. Dependable Computing and Fault-Tolerant Systems. Springer Verlag, Dec 1991.

Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996.

Evan Marcus and Hal Stern. *Blueprints for High Availability*. John Wiley & Sons, Inc., New York, NY, 2nd edition, 2003.

Daniel P. Siewiorek and Robert S. Swarz. *Reliable Computer Systems: Design and Evaluation*. AK Peters, Ltd., Natick, MA, 3rd edition, 1998.

Jan Van der Spiegel, James Tau, Titi Alailima, and Lin Ping Ang. The ENIAC – history, operation and reconstruction in VLSI. In Raúl Rojas and Ulf Hashagen, editors, *The First Computers – History and Architectures*. MIT Press, 2000.

