The Basics of Dependability

PODS (Fall 2007)

George Candea

EPFL / Dependable Systems Lab

George.Candea@epfl.ch

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. **?** 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, not security as defined here.

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. 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. Some authors (e.g., **??**) even replace MTTF with MTBF in the MTTF-based formula above, since MTTF and MTBF are often very close (because they are disproportionately greater than MTTR).

Similar metrics can be conceived for both safety and security, but they are generally less meaningful. For example, some literature uses the notion of mean-time-to-catastrophic-failure, often quoted as the expected value of $S(t)$, the probability that no catastrophic failure occurs during $[0, t)$. Since catastrophic failures constitute a subset of a system's failures, $S(t) \geq Rel(t)$ at all times. When discussing reliability, we refer to *how often* failures occur, whereas when we discuss safety it is more about *whether* catastrophic failures can occur—after one catastrophe occurs, there is little opportunity for that same system to induce another one.

In the case of security, various attempts have been made at trying to quantify it, and we will see some of these attempts later in the course. An interesting metric, mean-time-between-attacks, is calculated by the SANS Institute: their Internet Storm Center (ISC) listens to network traffic and tallies the number of attack probes run against IP addresses in various ISPs. If you assume that an unpatched system reached by such a probes would be infected, then that is the point at which the system will have witnessed a security failure. The mean-time-between-attacks would therefore be a measure of the average "lifespan" of an unpatched, unprotected PC connected to the Internet. **?** currently reports the mean-time-between-attacks to be 21 minutes, which is shorter than the time it takes to download and install critical patches for popular operating systems.

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: if the system fails before completion of a task, and the task has to be restarted anew, then little progress can be made.

The key to keep the ENIAC running was to reduce repair time (MTTR): a staff of 5 people patrolled the innards of the 30-ton ENIAC monster on roller skates and replaced tubes as soon as they blew out. Recently, **?** 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 **?** for a thorough treatment of reliability in computer systems.

Hardware reliability improved significantly over the two decades following ENIAC, paving the way for higher expectations. 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.

But then, 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 for answering 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. **?** is a good reference for building highly available systems.

As MTTF and MTTR of hardware and software improved, the availability of interactive systems reached a threshold of acceptability, 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 such applications, safety became an important issue, because many of these system's failures could lead to loss of human life, which is a highly catastrophic event.

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 **?**, the Patriots used during the First 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 hit US military barracks in Saudi Arabia, killing 28 soldiers and wounding 98. See **?** for detailed analyses of software systems and their role in the safety of larger industrial and military systems.

The last two decades have seen a tremendous growth in distributed systems and the Internet, largely driven by the Web and peer-to-peer applications. 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,

[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.

accompanied by information theft, has made the topic of computer security a high priority topic, and continues to be so today. New applications, like Internet telephony and e-commerce, pose the challenge of protecting data privacy as it traverses potentially-unfriendly networks. A system that is unable to prevent unauthorized access to data is not secure.

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 at several companies and agencies, 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. Since then, many cases of unauthorized data access have caused Internet services and software companies to invest heavily in data protection measures. For those with deeper interest in this topic, **?** constitutes a great treatise on the design of secure computer systems.

3. A Note on Correctness and Metrics

Solid engineering is predicated on the ability to make quantitative measurements so that successive improvements to a system can be evaluated and demonstrated. To make measurements, we must precisely define what *correctness* entails, i.e., when is a computer system's behavior considered correct vs. faulty.

Unfortunately, this is still a challenge. A number of attempts have been made to formalize the specification of computer systems, especially in the safety-critical industries (aerospace, automotive, medical devices, etc.) and in academic research circles. The more formal a specification becomes, the more difficult and expensive it is to generate. We will study a number of successful as well as failed examples in the course of this semester.

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 (these parameters define "correct service"). 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. Availability here means that the network delivers packets within prescribed time bounds. 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.

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. The curious can skip ahead and read **?** for a discussion of these problems.

4. Costs

Service level agreements will often stipulate penalties, but it is generally accepted that the financial losses resulting from potential shortcomings in reliability, availability, safety, and security are difficult to quantify. It is generally easier to perform an a posteriori analysis. For example at eBay, a popular online auction service, a flaw in the Sun Solaris operating system led to database file corruptions that brought the service down for 22 hours in June 1999; eBay's direct costs were estimated at $3-5 million, and eBay's stock price dropped by 26%, erasing $4 billion in market capitalization **?**. It is hard to believe that one could have estimated ahead of time that an unknown bug in Solaris's file system code would have these consequences. As another example, a bug in General Electric's XA/21 energy management system contributed in August 2003 to the scope of the worst power outage in U.S. history, that spread across the Northeastern U.S. and Canada; according to **?**, this outage affected 50 million people and disrupted transportation, water, and many other services. The total cost, however, was difficult

Industry / Company	Cost per hour
Brokerage operations	$6,450,000
Credit card authorization	$2,600,000
eBay	$225,000
Amazon.com	$180,000
Package shipping services	$150,000
Home shopping channel	$113,000
Catalog sales center	$90,000
Airline reservation center	$89,000
Cellular service activation	$41,000
On-line network fees	$25,000
ATM service fees	$14,000

Table 1. Cost of one hour of downtime (from ?)

to estimate.

Table 4 shows a few example estimates of average costs of downtime, broken down by industry/company. While these are not to be taken as a rule-of-thumb, they do convey the general order of magnitude.

In terms of global costs, a number of estimates have been made, to varying degrees of accuracy. In 2002, the U.S. National Institute of Standards and Technology (**?**) estimated that software faults cost the U.S. economy $59.5 billion the previous year, which represented 0.6% of the country's Gross Domestic Product. The Sustainable Computing Consortium estimates that software defects cost global business an estimated $175 billion in 2001. These numbers are hard to validate, so they should be taken with a grain of salt.

5. Classifications

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 and observed system behavior. In particular, the notions of fault, failure, and error are generally used loosely, yet they do have precise meanings.

5.1 Faults, Errors, and Failures

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

and would impact reliability. 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.

Another example illustrates how security, availability, and reliability intermix in such a propagation sequence. By exploiting a buffer overflow bug (fault) in Microsoft SQL Server, the SQL Slammer worm was able to overwrite victims' stacks with a copy of the worm (error), allowing the worm to execute and propagate to other machines. This caused infected SQL Server machines to generate rapidly-increasing level of network traffic, which was not part of their specification (failure). Under such high traffic load, network routers should normally delay delivery of packets, but instead some of them crashed, becoming unavailable to neighbor routers, which in turn removed the crashed neighbors from their routing tables and notified other peer routers about the failed routers. The flood of routing table updates caused more routers all over the world to fail; when network administrators restarted some of the failed routers, an additional compounding flurry of routing table updates ensued, confusing routers and causing

congestion in large parts of the Internet, preventing networks from delivering packets (failure). The official South Korean news agency reported that Internet services had been shut down for several hours in South Korea due to SQL Slammer.

5.2 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 to find the one, 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. According to Jim Gray, the term was coined by Bruce Lindsay, who one late night during the days of CAL TSS at UC Berkeley (1960's) was struck by the similarities between Heisenberg's Uncertainty Principle[2] and the bug behavior he was witnessing: the closer Bruce was getting to determining the location of the bug, the more erratically 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 paradoxical 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 now-regular 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 truly Schrödingbugs is left up to the reader to decide.

[2]Heisenberg's Uncertainty Principle is a cornerstone of quantum mechanics; it states that, even if using an infinitely precise instrument, one cannot measure both the position and the momentum of a physical object with arbitrary accuracy. In particular, the product of the uncertainties in position and momentum is equal to or greater than half of Planck's (reduced) constant. This principle was discovered by Werner Heisenberg in 1927.

Finally, a bug whose underlying cause is so complex and obscure, that it appears to be 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. A well-publicized example is the Pentium floating point bug: an older version of the Pentium microprocessor could cause certain divisions to be erroneous (essentially, the CPU had a faulty quotient digit selection table); this 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. Moreover, as the level of concurrency increases in software products (e.g., high-capacity servers), complex race conditions result in the manifestation of Mandelbugs.

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. Similarly, most safety and security failures tend to be permanent: if a plane has crashed or an intruder has stolen credit card accounts, it is difficult to recover from the failure. 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—these combined conditions occur only rarely. 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[3]; this is by far the "best" kind of system, because fail-stop behavior is relatively easy to deal with.

By contrast, **Byzantine** systems do not stop once they fail, rather they return wrong information. The term was first used by **?** 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

[3]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.

failures are the most difficult category of failures to deal with; to make an analogy to everyday life, think about dealing with liars versus dealing with people who can only say the truth or shut up. Unfortunately, Byzantine failures are quite frequent. E.g., any time a server returns corrupt data to a client, it is a Byzantine failure. If a program confirms it has committed a certain datum to storage, but due to a race condition it has actually failed to do so, it exhibits a Byzantine failure. Some of the most pernicious Byzantine failures result from malicious activity: when a bot compromises a PC and installs spyware, it will hide itself by replacing sytem binaries and libraries with hacked versions. When the system's users list the currently running processes, they do not see the spyware or bot processes—the system exhibits Byzantine behavior and, as a result, the spyware infection persists and can comfortable spread to other PCs. **?** provides a voluminous set of elegant algorithms to deal with Byzantine nodes in distributed systems; **?** 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. Often, the key to fail-fast behavior is fast detection.

5.3 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. This sequence constitutes a general model for how failures are handled.

Fault detection is the process of discovering the presence of a fault, usually by detecting the error that fault created or the failure that it causes. 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). Failure detection can often be used interchangeably with fault detection, since it is most often the case that we detect the failure (effect of a fault) rather than the fault itself. An example of failure detection is supplied by intrusion detection systems, which monitor systems to detect when their security perimeters have been penetrated. The most sophisticated detection systems construct a mathematical model of some aspect of the system's normal behavior (e.g., of network ports used by legitimate applications) and deviations from this model are flagged as anomalies, which may signal an intrusion; when a probable intrusion is detected, relevant information gets logged and various forms of alerts are generated.

Fault diagnosis is the next step, by which the detected error or failure, 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[4] and provide straightforward detection and diagnosis, that allows operators or software to fix the faulty modules. When a Web service is not reachable, usually the first step in diagnosis is to determine whether the Web server machine is reachable at the IP layer, if yes then whether the Web server process is still running, and so on; the analysis could yield a diagnosis such as "Apache daemon crashed."

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 to higher layers of the system. An alternative way is to a priori put up "walls" between parts of the system, to prevent potential faults from propagating: isolate programs onto separate hardware, run them inside virtual machines, etc. Such "walls" can also be put up dynamically; e.g., when an intrusion is detected or compromised PCs are found on a network, system administrators will typically first disconnect the PC from the network, to prevent it from spreading the attack further.

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. In the case of the crashed Apache server, it might be due to the machine having run out of disk space in `/tmp`, so the administrator will free up some space, so that Apache has where to store its temporary data.

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. Recovery almost always entails either repairing data ("hard state") or restarting components that only

[4]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.

handle volatile state (such as the Apache server).

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 deprives the fault of the time needed 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 one of the pillars of research done in the Dependable Systems Lab here at EPFL; we will survey some of these research ideas in future lectures.

6. A Look Ahead

Before studying dependability, we must first understand why computer systems fail, so in the next lecture we will analyze various studies of computer failures and draw conclusions about sources of failure and historical trends in the root causes of failure. We will then turn our attention to metrics and see how we can perform direct measurement through statistics, monitoring frameworks, client surveys, etc.

The second section of the course will focus on "tried and true" approaches to dependability that constitute the foundations of dependable system design. We will start with various techniques for achieving reliability and analyze their benefits and drawbacks. Then we look at availability and discuss widely used approaches like replication and redundancy in clusters along with fast recovery. For safety-critical systems, we will see various approaches to proving correctness and ensuring data safety and disaster recovery. In the context of security, we will discuss methods for preserving privacy and trust, for performing authentication and authorization, as well as keeping audit trails for compliance verification.

The third major section of the course is aimed at developing an understanding of how dependable systems are developed in industry. We will spend an entire lecture focusing on system manageability, user-centric design, and configuration management. We will explore the various techniques used for quality assurance during the typical software lifecycle, including testing, fault injection, robustness testing, and audit code.

The final part of the course is more heavily weighted on research aspects surrounding dependability. We will study systems for checkpointing and restart-based recovery and look at how to build adaptable systems exhibiting graceful degradation. In this context, we will identify a few successful principles and also look at recent advances

in the application of control theory and market models to the field of dependability. An entire lecture is dedicated to the topic of detection, diagnosis, and prognosis; here we will also learn about anomaly detection systems and various wide-area monitoring infrastructures. We will conclude the semester with material on software-based fault isolation, software rejuvenation, distributed consensus and agreement, and software predictability.

Welcome to the course and I hope you enjoy the ride!

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.

Jim Gray. Why do computers stop and what can be done about it? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, 1986.

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.

NIST. National Institute of Standards and Technology, May 2002. The economic impacts of inadequate infrastructure for software testing, Gaithersburg, MD.

David A. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of the 16th USENIX Systems Administration Conference*, Philadelphia, PA, 2002.

Kevin Poulsen. Software bug contributed to blackout. http://www.securityfocus.com/news/8016, February 2004. SecurityFocus.

SANS. The SANS Institute's Internet Storm Center. http://isc.sans.org/, 2006.

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

US GAO. 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.

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.

