

# An Introduction To Computing System Dependability

John C. Knight

*Department of Computer Science, University of Virginia*

*Charlottesville, VA 22904-4740*

*+1 434 982 2216*

*knight@cs.virginia.edu*

## 1. Introduction

Computer systems provide us with a wide range of services upon which we have come to depend. Many computers are embedded in more complex devices that we use such as automobiles, aircraft, appliances, and medical devices. Others are part of sophisticated systems that provide us with a variety of important facilities such as financial services, telecommunications, transportation systems, and energy production and distribution networks. Engineering systems to be as dependable as we require them to be is a significant challenge and requires a variety of analysis and development techniques.

It is important that computer engineers, software engineers, project managers, and users understand the major elements of current technology in the field of dependability, yet this material tends to be unfamiliar to researchers and practitioners alike. Researchers are often concerned in one way or another with some aspect of what is mistakenly called software “reliability”. All practitioners are concerned with the “reliability” of the software that they produce but researchers and practitioners tend not to understand fully the broader impact of their work. A lot of research, such as that on testing, is concerned directly with software dependability. Understanding dependability more fully allows researchers to be more effective. Similarly, practitioners can direct their efforts during development more effectively if they have a better understanding of dependability.

## 2. Concepts and Definitions

Many systems need to be *dependable*. The notion of dependability is broken down into six fundamental properties: (1) reliability; (2) availability; (3) safety; (4) confidentiality; (5) integrity; and (6) maintainability. Informally, it is expected that a dependable system will be operational when needed (availability), that the system will keep operating correctly while being used (reliability), that there will be no unauthorized disclosure (confidentiality) or modification (integrity) of information that the system is using, and that operation of the system will

not be dangerous (safety). Precise definitions of these terms have been developed over a period of many years by a group of researchers led by J.C. Laprie [2].

It is important to note that security [1] is not included explicitly in this list. Rather, it is covered by the second, fourth and fifth items in the list. It is also important to note that some aspects of dependability are not completely intuitive. For example: (1) a communications system can be highly available but unreliable; (2) an aircraft system can have low availability but be safe; and (3) an information system can be reliable yet insecure.

The first step in developing a dependable computer system is to define the dependability requirements. As an example, consider an implantable heart pacemaker. Clearly, such a device has to be dependable because a pacemaker failure is obviously serious. A pacemaker does not have any requirement for security but has very strong requirements for safety and availability. The device does not necessarily have to be reliable (in the formal sense). If it failed just after one heartbeat but reset itself and resumed proper operation before the next one, there would probably be no damage.

Once the dependability requirements for a system have been determined, the next step is to determine the system hazards. *Hazards* are system states that could lead to undesirable outcomes which result from using a system. Hazards are not obvious, and determining the hazards for a given system is often a difficult and complex task. The primary danger in most medical systems, for example, is patient injury, and so any state in which a patient might be injured is a hazard. For a pacemaker, the most serious and obvious hazard is failure to pace the patient's heart when required. A second, much less obvious hazard would be to pace the heart when it was *not* required. Either could be fatal.

If a system were perfect and stayed that way, it would never fail to supply acceptable service, and there would be no concern for its performance. However, systems do fail, they do break as time passes, and in some cases, they contain defects in their basic design. Informally, a *failure* has occurred when a system no longer meets users' expectations. The notion of failure is only relevant to what the system's users experience. This is an important concept

because the failure of components within a system is both expected and possibly common. The trick to dependable design is to make sure that failed components within a system do not lead to system failure.

An *error* is a system state which, if nothing is done to prevent it, might lead to a system failure. Thus, for example, a system state in which various data elements are corrupted for some reason might well lead to a system failure unless the system detects and corrects the error. It is important to remember, therefore that *failure* is when an *error* reaches the service interface. Errors that do not reach the service interface are not relevant to the user because they do not impact the service provided.

A *fault* is the adjudged cause of an error. A defect in a piece of software that causes some of the system's data elements to become corrupted is a fault. Similarly, computer hardware that malfunctions is a fault, as is a break in a power or communications cable caused by corrosion.

There are two basic types of fault—*degradation* faults and *design* faults. Degradation faults arise over time, and a component that operates correctly for some period of time might stop doing so as a result of a degradation fault. The simplest example of a degradation fault is when the filament in a light bulb breaks. Degradation faults have to be expected, and, if such faults could lead to failure, the system has to be designed to operate acceptably after such faults have occurred. This concept is referred to as *fault tolerance*. Thus fault tolerance is a mechanism by which, in part, a system dependability property (such as availability) is achieved.

A design fault is something that is wrong with a system that has been there from the outset. It is not the result of a component wearing out. All software faults are design faults because software does not degenerate in the usual sense. Design faults can be *avoided* altogether by careful design processes, *removed* during development by correcting the design after some form of analysis reveals the fault, or tolerated during operation.

To build a pacemaker that meets typical dependability requirements, it is necessary to deal with a variety of both design and degradation faults. Such a device is very complex and contains a lot of software. Thus a wide range of design faults must be expected including design faults in the microprocessor's circuitry, in the wiring between discrete components, and in the software. Similarly, a wide range of degradation faults must be expected including gate failures in the microprocessor, battery failure, sensor failure, and wiring failures between discrete components.

### 3. Software's Role

Software is a key component in any computer system, and it has been a factor in many serious failures. Numerous problems arise because of the interaction between software engineering and systems engineering. In a system that employs fault tolerance, it is usually the case that software implements the fault tolerance designed to deal with

hardware faults. In a pacemaker, for example, it is completely unacceptable for the device to cease operation if a sensor fails. Where the sensor is providing critical data about heart activity, it might be impossible to continue to provide full functionality without that sensor. The software must be written in such a case to diagnose the situation, record the details for subsequent analysis, and switch to a simple but safe emergency pacing algorithm.

The Achilles' heel of software development for most types of system is requirements specification. In the pacemaker example, it is necessary for the software specification to include full medical functionality, hardware fault diagnosis, patient data recording, communication with external equipment, and emergency functionality. Specification of such systems is difficult and error prone.

A great deal of effort has been expended on the development of formal notations for specification and analysis with the goal of allowing complete, unambiguous and analyzable specifications to be built. There have been notable successes with this technology.

Verification is another major challenge in the development of software for most applications. Again, many techniques have been developed to facilitate verification including formal verification and model checking. Testing, however, remains the dominant approach to verification in practice. Testing also remains problematic. It is impossible to execute a sufficient number of tests to permit a statistical assessment of extreme dependability and so the results of almost any testing process are informal [3]. Running tests on many systems is difficult because test harnesses have to duplicate a complex operating environment.

## 4. Conclusion

Computer systems offer tremendous capabilities but making them dependable is a complex process. An elaborate interplay between techniques from systems engineering and software engineering has to be undertaken. Additional information can be obtained from textbooks such as those by Storey [5] and Leveson [4], from a variety of journals and conferences.

## 5. References

- [1] Anderson, R., *Security Engineering: A Guide to Building Dependable Distributed Systems*, Wiley, 2001.
- [2] Avizienis, Laprie, Randell, "Fundamental Concepts in System Dependability", *IARP / IEEE-RAS Workshop on Robot Dependability*, May 2001.
- [3] Butler, R. and G. Finelli, "The infeasibility of experimental quantification of life-critical software reliability." *IEEE Trans. on Software Engineering*, 19(1):3--12, Jan. 1993.
- [4] Leveson, N., *Safeware: System Safety and Computers*, Addison Wesley, 1995.
- [5] Storey, N., *Safety-Critical Computer Systems*, Addison Wesley, 1996.