African University of Science and Technology

# Fault tolerance

Pr. Ousmane THIARE

http://www.ousmanethiare.com

August 18, 2014

## Outline

## Introduction to Fault tolerance

Fault tolerance is subject to much research in computer science. In this section, we start with presenting the basic concepts related to processing failures, followed by a discussion of failure models. The key technique for handling failures is redundancy, which is also discussed. For more general information on fault tolerance in distributed systems, see, for example (Jalote, 1994).

## Outline

## Basic concepts

To understand the role of fault tolerance in distributed systems we first need to take a closer look at what it actually means for a distributed system to tolerate faults. Being fault tolerant is strongly related to what are called dependable systems. Dependability is a term that covers a number of useful requirements for distributed systems including the following (Kopetz and Verissimo, 1993):

- Availability
- Reliability
- Safety
- Maintainability

## Availability

### Definition

**Availability** is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users. In other words, a highly available system is one that will most likely be working at a given instant in time.

## Relability

### Definition

**Reliability** refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly reliable system is one that will most likely continue to work without interruption during a relatively long period of time. This is a subtle but important difference when compared to availability. If a system goes down for one millisecond every hour, it has an availability of over 99.9999 percent, but is still highly unreliable. Similarly, a system that never crashes but is shut down for two weeks every August has high reliability but only 96 percent availability. The two are not the same.

## Safety

### Definition

**Safety** refers to the situation that when a system temporarily fails to operate correctly, nothing catastrophic happens. For example, many process control systems, such as those used for controlling nuclear power plants or sending people into space, are required to provide a high degree of safety. If such control systems temporarily fail for only a very brief moment, the effects could be disastrous.
Many examples from the past (and probably many more yet to come) show how hard it is to build safe systems.

## Maintainability

### Definition

Finally **maintainability** refers to how easy a failed system can be repaired. A highly maintainable system may also show a high degree of availability, especially if failures can be detected and repaired automatically. However, as we shall see later in this chapter, automatically recovering from failures is easier said than done.

A system is said to **fail** when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided. An **error** is a part of a system's state that may lead to a failure. For example, when transmitting packets across a network, it is to be expected that some packets have been damaged when they arrive at the receiver.

## Outline

## Faults Classification

Faults are generally classified as transient, intermittent, or permanent. **Transient faults** occur once and then disappear. If the operation is repeated, the fault goes away.
An **intermittent fault** occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, whenever the fault doctor shows up, the system works fine.
A **permanent fault** is one that continues to exist until the faulty component is repaired. Burnt-out chips, software bugs, and disk head crashes are examples of permanent faults.

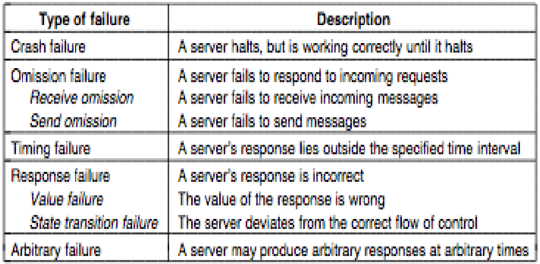
## Outline

## Failure Models

A system that fails is not adequately providing the services it was designed for. If we consider a distributed system as a collection of servers that communicate with each other and with their clients, not adequately providing services means that servers, communication channels, or possibly both, are not doing what they are supposed to do. However, a malfunctioning server itself may not always be the fault we are looking for. If such a server depends on other servers to adequately provide its services, the cause of an error may need to be searched for somewhere else.

## Failure Models

To get a better grasp on how serious a failure actually is, several classification schemes have been developed. One such scheme is shown in the figure below, and is based on schemes described in (Cristian, 1991; and Hadzilacos and Toueg, 1993).



| Type of failure | Description |
|---|---|
| Crash failure | A server halts, but is working correctly until it halts |
| Omission failure | A server fails to respond to incoming requests |
| *Receive omission* | A server fails to receive incoming messages |
| *Send omission* | A server fails to send messages |
| Timing failure | A server's response lies outside the specified time interval |
| Response failure | A server's response is incorrect |
| *Value failure* | The value of the response is wrong |
| *State transition failure* | The server deviates from the correct flow of control |
| Arbitrary failure | A server may produce arbitrary responses at arbitrary times |

Figure : Different types of failures.

## Failure Models

A **crash failure** occurs when a server prematurely halts, but was working correctly until it stopped. An important aspect with crash failures is that once the server has halted, nothing is heard from it anymore. A typical example of a crash failure is an operating system that comes to a grinding halt, and for which there is only one solution: reboot.
An **omission failure** occurs when a server fails to respond to a request. Several things might go wrong. In the case of a receive omission failure, the server perhaps never got the request in the first place.
Another class of failures is related to timing. **Timing failures** occur when the response lies outside a specified real-time interval.

## Failure Models

A serious type of failure is a **response failure**, by which the server's response is simply incorrect. Two kinds of response failures may happen. In the case of a value failure, a server provides the wrong reply to a request. For example, a search engine that systematically returns Web pages not related to any of the used search terms, has failed.
The other type of response failure is known as a **state transition failure**. This kind of failure happens when the server reacts unexpectedly to an incoming request. For example, if a server receives a message it cannot recognize, a state transition failure happens if no measures have been taken to handle such messages.

## Failure Models

The most serious are **arbitrary failures**, also known as **Byzantine failures**. In effect, when arbitrary failures occur, clients should be prepared for the worst. In particular, it may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect.

## Outline

## Failure Masking by Redundancy

If a system is to be fault tolerant, the best it can do is to try to hide the occurrence of failures from other processes. The key technique for masking faults is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy (see also Johnson, 1995).
With time redundancy, an action is performed, and then, if need be, it is per- formed again. With physical redundancy, extra equipment or processes are added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. Physical redundancy can thus be done either in hardware or in software.

## Replication of Data

**Goal:** - maintaining copies on multiple computers (e.g. DNS)

**Requirements**

- Replication transparency – clients unaware of multiple copies
- Consistency of copies

**Benefits**

- Performance enhancement
- Reliability enhancement
- Data closer to client
- Share workload
- Increased availability
- Increased fault tolerance

## Replication of Data

**Constraints:**

- How to keep data consistency (need to ensure a satisfactorily consistent image for clients)
- Where to place replicas and how updates are propagated
- Scalability

## Recovery

- Once failure has occurred in many cases it is important to recover critical processes to a known state in order to resume processing
- Problem is compounded in distributed systems

**Two approaches**

- **Backward recovery**, by use of checkpointing (global snapshot of distributed system status) to record the system state but checkpointing is costly (performance degradation)
- **Forward recovery**, attempt to bring system to a new stable state from which it is possible to proceed (applied in situations where the nature if errors is known and a reset can be applied)