Lecture notes: About failures in the Cloud

M2 MOSIG: Software Infrastructures for Data Centers and Clouds

Thomas Ropars

2019

1 Some definitions

1.1 Fault, error, and failures

An **error** is the part of the system state that may cause a subsequent **failure**: a failure occurs when an error reaches the service interface and alters the service. A **fault** is the adjudged or hypothesized cause of an **error**

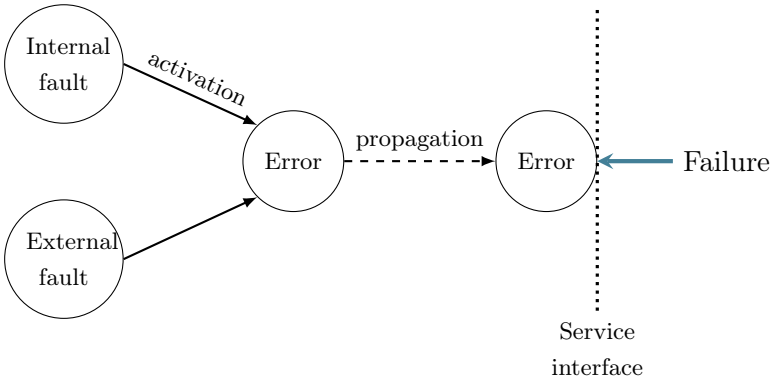

Figure 1: Fault/error/failure

The failure of a component is a fault from the perspective of the component using it, as illustrated in Figure 2.

Furthermore, we can distinguish different kinds of failures:

- **Catastrophic failures**: Large impact on the users and/or the environment. This is the kind of failures we want to avoid (other failures are called minor failures).
- **No response**: The system does not respond anymore.
- **Data failures**: The output is provided but it is a wrong value.
- **Timing failures**: System returns a good behavior but out of the timing expectations

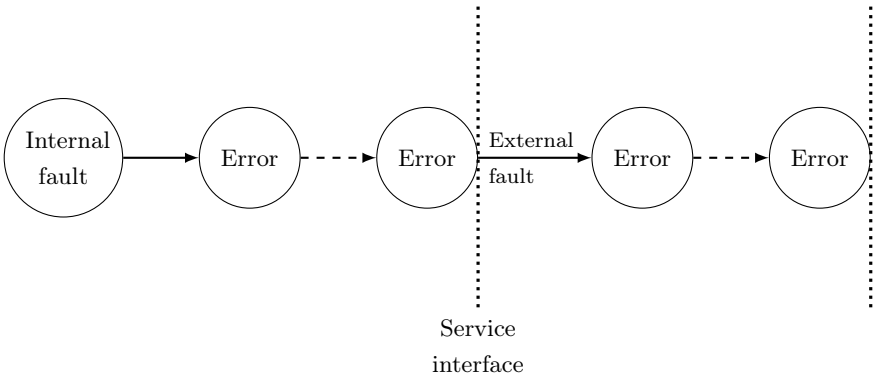

Figure 2: About errors and failures

- A system with timing constraints is called *real-time*.

- **Random failures (Byzantines)** - The system provides *random* outputs to the same inputs
  - Different users might observe different behaviors
  - Values might not be that random (malicious behavior)

1.2 About dependability, reliability and availability

Dependability is a general term that defines to which extend a service is able to avoid failures. Dependability encompasses several concepts:

- **Reliability**: The ability of a component to perform its required functions for a specified period of time
- **Availability**: The degree to which a component is operational and accessible when required for use
- **Safety**: Absence of catastrophic consequences on the user and environment
- **Confidentiality**: Absence of unauthorized disclosure of information
- **Integrity**: Absence of improper system state alteration
- **Maintainability**: Ease of repairing a failed system

Security is another important concept. It can be seen as a combination of **Confidentiality**, **Integrity** (the prevention of the unauthorized modification or deletion of information), and **Availability** (the prevention of the unauthorized withholding of information).

1.3 A focus on availability

Availability can be defined as the degree to which a component is operational and accessible when required for use. Said differently it is the portion of time for which a component is able to perform its function.

To compute availability, we need to introduce two parameters: MTBF (Mean Time Between Failures), MTTR (Mean Time To Repair). Hence we can compute the availability of a system as follows:

$$A = \frac{Uptime}{Uptime + Downtime} = \frac{MTBF}{MTBF + MTTR}$$

Table 1 presents the concept of high availability (*five nines*). Mission critical systems should be *ultra available*. For services in distributed systems, *high availability* is often the target.

System type	Unavailability (min/year)	Availability
Managed	5256	99 %
Fault tolerant	53	99.99 %
Highly available	5	99.999 %
Ultra available	0.05	99.99999 %

Table 1: About nines classes

2 Failures in Data Centers and Clouds

We start this section with an overview of the kind of failures that one may have to deal with in the cloud.

- Hardware failures
  - Each component of a distributed system has a very high MTBF (more than 10 years for HDDs). However, with a very high number of components, the risk of experiencing a failure becomes much higher.
  - Very often we run in virtual machines. These machines may be interrupted, especially in the case of preemptible instances. From the application point of view, this may be equivalent to a hardware failure.
- Software failures
  - Any software component in a system includes bugs. Hardware failures may also lead to software failures.
  - Software failures often result in correlated failures (all replicas of a service fail at the same time).
  - Note that automatic testing can reveal many software bugs.

- Comment on human errors
  - Platforms and software are designed and operated by humans. Humans make mistakes.
  - Many services outage comes from human errors. The human errors will translate into software or hardware failures.
  - Solutions to limit the human errors: automatizing (infrastructure-as-code), devops, etc.

Dealing with failures in a data center or in a cloud is difficult for many reasons (and probably more difficult than in a supercomputer):

- A large majority of applications are online services that we cannot afford to stop and restart. We need to target *high availability*.
- Failures are frequent because these systems might be built out of cheaper hardware (compared to supercomputers)

At the end, we can't assume that all faults will be managed directly by the hardware. Fault tolerance mechanisms have to be built at the software level.

3 A closer look at failures in the Cloud

3.1 The occurrence of failures in Clouds

Dealing with failures requires to be able to detect failures. Some fault tolerant systems may continue working despite hardware or software failures that would impact a subset of the components of the system. However, ultimately failed components need to be replaced to maintain the same level of availability for the services.

On a single machine, it is most of the time easy to know if everything works correctly or not, especially because most errors lead to a crash. In a distributed system, things can be more complex.

Definition 1 (Distributed systems – by L. Lamport) *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.*

Detecting failures in distributed systems is difficult for two main reasons:

Partial failures: Some parts of the system may stop working in unpredictable ways (but the other parts of the system might not be aware of it)

Asynchronous system: The time for messages to travel across the network may significantly vary.

When discussing about data centers and clouds, we are considering mostly **shared-nothing** infrastructures. It means that the only way to interact between the nodes is by sending/receiving messages over the network.

The network in a data center should be considered as unreliable and asynchronous. When a packet is sent, it is not guaranteed that it will arrive, and we don't know when it will arrive.

In such a system, discovering if something is going wrong and what is going wrong is difficult. Imagine that a node `A` sends a request to another node `B` and does not receive an answer. What could be the reasons?

- The request was lost and never reached `B`.
- The request is delayed and has not been delivered yet.
- Node `B` is down (maybe because it crashed or because it shut down)
- Node `B` may be experiencing a transient issue (slow-down) and will respond later
- Node `B` may have processed the request but the answer was lost on the way back
- Node `B` may have processed the request but the delivery of the answer is delayed

At the end, it is impossible to tell why no answer was received. It is not even possible to know whether the request has been processed or not. All of these reasons make it challenging to implement fault tolerance in this kind of infrastructure.

It should be mentioned that there multiple examples that evidence the fact that networks should not be considered as reliable and that network partitioning is something that can occur. Read "*The network is unreliable*" by Bailis and Kingsbury for a detailed discussion on this topic. The following is extracted from this article. Google Fellow Jeff Dean suggested that a typical first year for a new Google cluster involves:

- Five racks going wonky (40-80 machines seeing 50 percent packet loss).
- Eight network maintenance events (four of which might cause 30-minute random connectivity losses).
- Three router failures (resulting in the need to pull traffic immediately for an hour).

It should also be mentioned that users of public cloud services such as EC2 may experience a non-negligible number of temporary network problems. Network partitioning between regions have been observed several times.

Finally, it appears that a non-negligible number of the network errors are due to human errors (e.g., configuration issues at the level of switches).

3.2 About network delays and unreliable communication

Even if the network is properly configured and does not experience hardware issues, the level of service that it provides might be low (packet switching).

There are several reasons why a packet delivery might be delayed. Basically, a packet is delayed because it is queued at some level of the network stack. Here are a few possible reasons:

- Multiple nodes trying to send to the same destination may force a network switch to buffer some packets while the link is busy. Ultimately, the switch may have to drop packets.
- If the destination service is running inside a VM, and it is sharing the hardware resources with other VMs, it might need to wait before getting access to the hardware.
- The destination service might just have too many requests to process.

All these issues exist in any packet-switching network, i.e. in most computer networks today.
In Cloud infrastructures, the issues are exacerbated by the fact that:

- All resources, including the network infrastructure, are shared with other users (and we may have *noisy* neighbors).
- Resources may be very close or far away (geo-distributed environments).

Hence, it is well documented that *jitter* can be large when observing response time in Cloud environments:

- Can be observed here for instance: `https://itm.cloud.com/google-reports/`
- Observation made on the GCE API on 11/18/19 – latency measures on my personal account:
  - Median latency: 295 ms
  - 90-percentile latency: up to 500 ms
  - 99-percentile latency: up to 1300 ms

3.3 Detecting failures using timeouts

It is sometimes possible to get direct feedback about failures:

- If a process crashes but the node is still alive, we may receive an error message when trying to establish a network connection.
- Again, if a node is alive, we may implement a low-level service that informs about the status of other services
- A network switch might return an error when a destination is unreachable.

Note that even if we are able to detect a failure, we might not know if a request or how much of a request was already processed by the destination.

To detect failures, we will mostly rely on timeouts: if a request is not acknowledged before a certain time, we consider that the communication failed. But setting up timeouts is also a challenging task:

- Having small timeouts allows reacting fast to failures. But wrongly suspecting a node may have a huge cost (reconfiguration of the service). It may result in some actions being performed more than once.
- Having large timeouts reduces the risk of false positives but users may experience delays in request processing.

There is no perfect answer to this problem. Note however that in general, a single policy will not fit all services. Even considering a single service/application, one may want to rely on different timeouts for different decisions. Take the example of a master-worker application:

- A small timeout might be used to detect temporarily slow workers, and avoid assigning new work to those
- A large timeout might be used to detect dead workers that should be replaced by others.

3.4 Fault model

To be able to reason about fault tolerance techniques, we need to define an abstract model that captures the main characteristics of the faults we want be able to tackle.

In the following, we will assume **benign** faults (as opposed to byzantine faults). For processes, benign faults correspond to crashes: a process either executes according to the specification, or stops executing. For channels, benign faults correspond to the loss of messages.

4 Introduction to replication

The basic way of interacting between services/processes in Clouds and data centers is through a client/server model. A client sends a request to another node and waits for an answer.

Replication is a technique that allows us to increase the availability of a system or service.

With replication, instead of having only one instance of a system/service, there are several copies. So, if one copy crashes, the system/service will still be available, thanks to the other copies (Fig. 3).

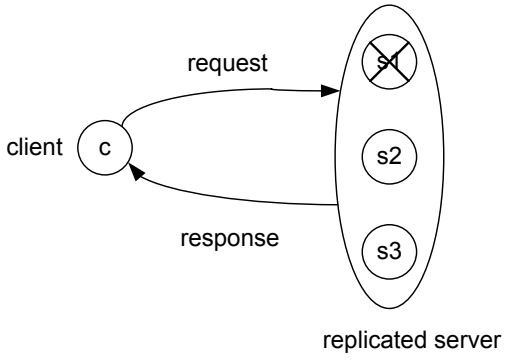

Figure 3: Replication (client-server model)

Several approaches to replication exists. We can distinguish 3 main categories:

Data replication : With this approach, the client can only issue read and write operations on the data. On the other hand, it allows a high degree of parallelism.

Passive replication : This approach is also called *active/standby* replication. A single leader executes all requests and sends updates to the other replicas.

Active replication : This approach is also called *active/active* replication, or multi-leader applications. In this approach, all replicas can process requests.

Note that availability might not be the only concern when using replication. Replication may also be used to improve performance:

- To reduce the latency by keeping data geographically close to the users
- To improve throughput by allowing parallel reads

These concerns should also be taken into account when choosing a replication technique.

Replication techniques will be discussed in details in the next lectures.

References

Some references can complement the material presented in these lecture notes:

- *Fundamental concepts of dependability* by Avizienis, Laprie, and Randell.
- Chapter 8 of *Designing Data-Intensive Applications* by Martin Kleppmann

Note that some parts of these lectures are also strongly inspired by the lectures notes of Andre Schiper on *Distributed Algorithms*.