Lecture notes: Studying distributed systems – About abstractions and failure detectors

M2 MOSIG: Large-Scale Data Management and Distributed Systems

Thomas Ropars

2024

1 Abstractions

Distributed systems are complex systems with a large diversity in hardware and software. Servers can vary in the number of processors, their type, the storage they have access to, etc. Communication infrastructures differ in performance, reliability, topology, etc. Hence, to be able to reason about distributed systems, we need to define abstractions.

The first risk, if we would focus on a specific distributed infrastructure, would be to design solutions that could only work on that specific infrastructure. The second risk is that the systems are so complex that trying to take into account all the problems at once would be so difficult that we would never reach to a solution.

We can note that in the previous lectures, we already used some abstractions: We represented the activity of a process as a sequence of events; To define the Chandy-Lamport snapshot algorithm, we assumed FIFO channels.

1.1 A component model

As in software engineering, we will think about the abstractions we define in terms of components (or modules) with an interface. To build complex systems, we will use a *stack* of components where a component higher in the stack will rely on the service provided by components below, as illustrated in Figure 1.

To specify a component, we first need to define its *interface* (API), that is, how the other components can interact with it. Then, we need to define its *properties*, that is, the service/guarantees it provides to other components. Finally, we have to provide an *algorithm* that implements the expected properties.

2 Fault models

One core challenge when designing and implementing distributed systems is the fact that different problems (called *faults*) can occur at the hardware and at the software level. A piece of hardware may stop working completely or partially. There can be bugs and misconfigurations. A server might stop responding because it is overloaded, etc.

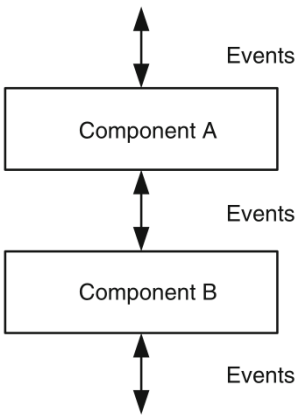


Figure 1: Composition model [1]

Once again, it would be too complex to reason about and handle each specific fault independently. Hence, we define fault models that capture the main consequences of different kind of faults. In a distributed system, faults might impact processes and communication links. Hence, we need to define fault models for each of them.

2.1 Fault model for processes

In our model, a process executes an algorithm implemented through a set of components. A failure occurs whenever the process does not behave according to the algorithm. The unit of failure is the process: When a process fails, all the components executed by the process fail.

We can consider different models depending on the fault that causes a process to fail. The most common fault model is *crash-stop*. This is the main model that we will consider during the course.

Crash-stop model This model can be described as follows. The process executes its algorithm until some time t after which it stops executing. We say that the process *crashes* at time t. When it crashes, the process stops executing any local computation and does not send any message to other processes The process does not recover after a crash.

In the crash-stop model, it is usual to distinguish between *correct* and *faulty* (i.e., *non-correct*) processes. The notion of "correct" refers to the whole execution of a process. A process is said to be *correct* if it does not crash; a process that crashes is said to be *faulty*.

Other models Other faults models can be considered for processes. The two main ones are the *crash-recovery* and the *byzantine* fault model:

- *Crash-recovery:* In this model, a process can crash and stop to send messages, but might recover later. At the moment, it resumes its execution, its state might not be up-to-date, since it crashed for some time. An algorithm designed based on this model needs to be able to deal with processes that would send outdated messages.
- *Byzantine faults:* Also called *Arbitrary fault*. This model is used to describe processes that may behave arbitrarily. The process may crash, drop messages, stop behaving according to

the algorithm it is supposed to execute, behave differently depending on the process it is exchanging messages with, etc. Among other things, this model is used to represent *malicious* behaviors.

Crash-stop vs crash-recovery Based on the provided definitions, we could wonder whether the crash-stop model can be useful in practice since it states that a process never recovers.

Actually, the crash-stop model does not prevent processes from recovering. However, an algorithm designed for the crash-stop model should continue working even if a crashed process never recovers. To deal with this issue, in the crash-stop model, we will assume that a majority of processes never crash.

2.2 Fault model for channels

With respect to links (channels) the following definitions can be considered:

- *Integrity*: A link from p to q satisfies integrity if q receives a message m at most once, and only if p previously sent m.

with one of the following properties:

- *Fair link*: A fair link from p to q satisfies integrity and the following property:
  If p sends infinitely many messages and q is correct, then q receives infinitely many messages.
- *Reliable link*: A reliable link from p to q satisfies integrity and the following property:
  If p executes $send(m)$ and q is correct, then q eventually receives m.
- *Quasi-reliable link*: A quasi-reliable link from p to q satisfies integrity and the following property:
  If p executes $send(m)$, and p and q are correct, then q eventually receives m.

Let process q be correct. Assuming reliable links, if p executes $send(m)$ at time t, and crashes at time $t+1$, then q must eventually receive m. This is not the case with quasi-reliable links, which define a weaker property. The reliable link definition does not adequately model existing transport layers (e.g., TCP). However, reliable links are useful to prove impossibility results (they strengthen impossibility results). Note that the two definitions are equivalent if processes do not crash.

2.3 Implementation of quasi-reliable links

In the following, we study how to implement *quasi-reliable* links on top of *fair* links. Through this implementation, we will illustrate the use of abstractions and how components can be combined to solve a problem.

We start by introducing the concept of *stubborn* links. A *stubborn* link satisfies integrity and the following property:

- *Stubborn link*: If a correct process p sends a message m once to a correct process q, then q delivers m an infinite number of times.

```
1   Implements:
2   StubbornPointToPointLinks, instance sl.

4   Uses:
5   FairPointToPointLinks, instance fl.

7   Variables:
8       sent = ∅

10  Init():
11      starttimer(Δ) # triggers a timeout after Δ time units.

13  Upon event Timeout:
14      forall (q,m) ∈ sent:
15          fl.send(q, m) # send msg m to process q using the fair link component
16      starttimer(Δ)

18  Upon sl.send(q, m):
19      fl.send(q, m)
20      sent = sent ∪ (q,m)

22  Upon fl.deliver(p,m)
23      Trigger sl.deliver(p, m)
```

Figure 2: Implementation of stubborn links: Retransmitting forever. p represents the local process and q the remote process

Algorithm for stubborn links We propose an algorithm that implements *stubborn* links in Figure 2. The algorithm is simple: For a given channel from process a to process b, it keeps retransmitting all messages sent by a to b. It assumes that a timer mechanism exists, that can trigger an action after a delay Δ.

Algorithm 2 is obviously not efficient from performance point of view. However, the first goal here is to understand how different components can be used together to provide a high-level service: our stubborn-link algorithm relies on a component implementing fair links (see line 15 for instance); later, we will rely on the stubborn-link component to implement quasi-reliable links.

Note that in the description of the algorithm, we use the word *deliver* instead of *receive*. We use *deliver* to refer to the moment when a component has decided, based on the algorithm it executes, to transmit a message to the component above. This can be different from the moment when the process has received the message (for instance if a FIFO property should be implemented).

To improve the performance of Algorithm 2, we could think of implementing an *acknowledgment* to prevent p from sending messages q has already received. However, if we would do so, the algorithm would not comply with the stubborn property, as defined above.

Algorithm for quasi-reliable links Figure 3 presents a solution to implement *quasi-reliable* links based on *stubborn* links. The idea is to eliminate the duplicates generated by the stubborn links.

In addition to the performance issues related to the use of stubborn links, the proposed algorithm introduces another problem related to the size of the `delivered` set that could become very large.

```
24      Implements:
25      QuasiReliablePointToPointLinks, instance qrl.

27      Uses:
28      StubbornPointToPointLinks, instance sl.

30      Variables:
31          delivered = ∅

33      Upon qrl.send(q, m):
34          sl.send(q, m)

36      Upon sl.deliver(p,m)
37          if m ∉ delivered:
38              delivered = delivered ∪ m
39              Trigger qrl.deliver(p, m)
```

Figure 3: Implementation of quasi-reliable links: Eliminating duplicates

A solution to this problem could be, once again, to use *acknowledgment* messages to notify the *source* process (P_a) that the *destination* process (P_b) has already delivered a given message m. When P_a stops sending a message, P_b could remove it from its `delivered` set. But it creates a new issue. What if an acknowledgment message gets delayed, and P_b deletes a message m_k from its set too early (*i.e.*, before the acknowledgment was received)? There is a chance that message m_k would be sent and delivered again, violating the *Integrity* property. Additional mechanisms based on timestamps could be used to fix the problem.

3 The synchronous system model

Until now, we have considered an asynchronous system model. An alternative is to consider a *synchronous* system model that can be defined as follows.

In a synchronous system there is (1) a known bound Δ on the transmission delay of messages, and (2) a known bound β on the relative speed of processes:

- *Bound on message delay:* If message m is sent by process p to process q at time t, then q receives the message no later than at time $t + \Delta$.[1]
- *Bound on the relative speed of processes:* If the fastest process takes x time units to do some computation, then the slowest process does not take more than $x\beta$ time units to do the same computation.

3.1 Synchronous model and failure detection

A difficult problem with the asynchronous system model is to detect failures: How to make the difference between a failed and a slow process?

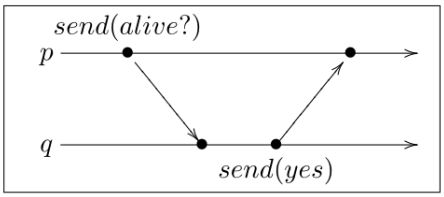
[1] In a synchronous system, channels are assumed to be reliable.

A synchronous system allows accurate failure detection:

$$process\ p\ crashes\ at\ time\ t \iff p\ is\ accused\ of\ having\ crashed\ at\ t' > t.$$

For example, for process p to know whether process q has crashed or not:

- p sends "*are you alive*" to q
- upon reception of this message, q sends "*yes*" to p



Process p knows that handling the "are you alive" message and sending the reply is done in x time unit by the fastest process. So if p has not received the "yes" message after $2\Delta + x\beta$, it knows that q has crashed.

3.2 Synchronous model in practice

Although the synchronous model simplifies the design of distributed algorithms, it has the major drawback that it is difficult to apply in practice. Building a distributed infrastructure with very low variations on the transmission delays and strong guarantees on the time it takes for a process to react to some event, can be difficult.

An alternative would be to set Δ and β to large (pessimistic) values to ensure that they always hold even if there are delays in the real system. However, a performance problem appears in this case. If an algorithm relies on these bounds to take decisions, it will become very slow.

Later in the course, we will introduce the *partially* synchronous model to overcome the limitations of the pessimistic approach imposed by the synchronous model.

4 Failure detectors

When defining a model for processes and channels earlier, we did not introduce a notion of time, as it is the case for the synchronous execution model. Indeed, in distributed systems timing assumptions are mostly used to detect failures.

To capture timing assumptions without having to redefine a process and a channel model, the concept of (unreliable) *failure detector* has been introduced by Chandra and Toueg [2]. A failure detector can be used to augment an asynchronous system and solve some problems that could not be solved otherwise (*e.g.*, consensus).

Failure detectors give each process an information — that can be unreliable — about the status crashed/alive of the other processes [2].

The failure detector model is defined as follows. Each process p_i has access to a local *failure detector module* FD_i that it can query (see Figure 4). Each local module FD_i monitors the processes in the system (or a subset of the processes), and maintains a list of processes that it currently suspects to have crashed.

Moreover, we have the following list of basic properties for failure detectors:

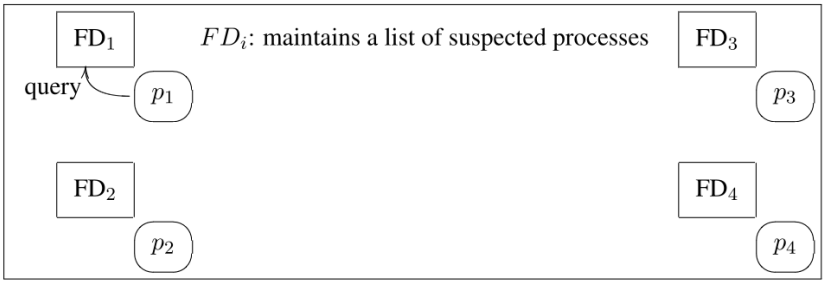


Figure 4: Failure detector module

- Each failure detector can make mistakes by erroneously adding processes to its list of suspects (i.e., it can suspect a process that has not crashed). In other words, the failure detectors are *unreliable*.
- A failure detector can change its mind by removing processes from its list, if it believes that the suspicion was erroneous.
- At a given time the failure detector modules at two different processes may have different lists of suspects.

If we do not set any constraint on the output of the failure detectors, it does not change anything compared to a simple asynchronous system. Hence, the idea is to add constraints, expressed in terms of two abstract properties: a *completeness* property and an *accuracy* property. Completeness sets constraints with respect to crashed processes, while accuracy sets constraints with respect to correct processes.

Completeness: Two completeness properties have been defined:

- *Strong Completeness*: Eventually every process that crashes is permanently suspected by *every* correct process.
- *Weak Completeness*: Eventually every process that crashes is permanently suspected by *some* correct process.

Accuracy: Four accuracy properties have been defined:

- *Strong accuracy*: No process is suspected before it crashes.
- *Weak accuracy*: Some correct process is never suspected.
- *Eventual strong accuracy*: There is a time after which correct processes are not suspected by any correct process.
- *Eventual weak accuracy*: There is a time after which some correct process is never suspected by any correct process.

Class of failure detectors: A pair (*completeness property, accuracy property*) defines a class of failure detectors. For example failure detector names have been attached to the following pairs:

- *Perfect* failure detector, denoted $\mathcal{P}$: satisfies strong completeness and strong accuracy.
- *Eventually perfect* failure detector, denoted $\Diamond\mathcal{P}$: satisfies strong completeness and eventual strong accuracy.
- *Strong* failure detector, denoted $\mathcal{S}$: satisfies strong completeness and weak accuracy.
- *Eventually strong* failure detector, denoted $\Diamond\mathcal{S}$: satisfies strong completeness and eventual weak accuracy.

$\mathcal{P}$ defines a property stronger than $\Diamond\mathcal{P}$, which defines a property stronger than $\Diamond\mathcal{S}$. $\mathcal{P}$ also defines a property stronger than $\mathcal{S}$.

The failure detector class $\mathcal{P}$ captures the ability to detect failures in a synchronous system.

We will study in upcoming lectures how failure detectors can be used to implement some distributed algorithms.

References

[1] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[2] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.