Abstractions

Etienne Renault

October 2, 2020

https://www.lrde.epita.fr/~renault/teaching/algorep/

Forewords

Success really depends on the conception of problems, the design of the system, not on the details of how it's coated.

Leslie Lamport

A need for abstractions

Studying distributed systems is not trivial and many factors have to be considered:

- Communication infrastructures: latency, throughput, reliability, protocols
- Operating systems: preemption, multithreading, Ghz, etc.
- Filesystem, middleware
- Failures

A need for abstractions

Studying distributed systems is not trivial and many factors have to be considered:

- Communication infrastructures: latency, throughput, reliability, protocols
- Operating systems: preemption, multithreading, Ghz, etc.
- Filesystem, middleware
- Failures

What about details?
What about complexity?

Table of Contents

Maximal Abstraction

A distributed system is composed of computational entities communicating by exchanging messages.

- Processes (computing elements)
- Links (network)
- Messages (data/information)

Processes (informally)

Associated to each node $i \in V$ we have a process.
A process is defined by a state machine, i.e. an algorithm.

Three kind of events can arise during a process'life:

1. **local event**: the status of the process has changed
2. **send message**: some information has been send to another process
3. **receive message**: some information has been received from another process

Distributed Systems (formally) 1/2

Consider $G = (V, E)$ a directed graph, with:

- V the nodes (*processes*) in the network
- E the edges (*channels*) of the network
- $\forall i \in V, out_i$ denotes the outgoing neighbours of node i,i.e. nodes from which there are edges from i in G.
- $\forall i \in V, in_i$ denotes the incoming neighbours of node i,i.e. nodes from which there are edges to i in G.

Transitions Functions

Suppose we have some fixed message alphabet M, we denote by:

- msg_i a message-generation function mapping $states_i \times out_i$ to elements of $M \cup \{null\}$
- $trans_i$ a state-transition function mapping $states_i$ and vectors of elements of $M \cup \{null\}$ (indexed by in_i) to $states_i$.

Associated with each edge (i, j) there is a channel which is just a location that can hold messages.

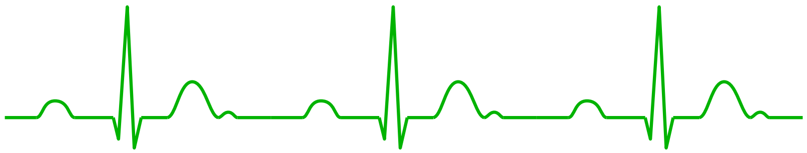
Various Timing models

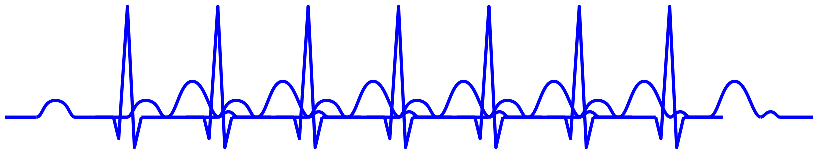
Synchronising processes is one of the most difficult part of distributed system

Multiple models exit:

- **Asynchronous model**: *Processes do not share a clock*
- **Synchronous model**: *Processes share a clock*
- **Partially-synchronous model**: *Processes have approximately synchronized clocks*

Asynchronous model



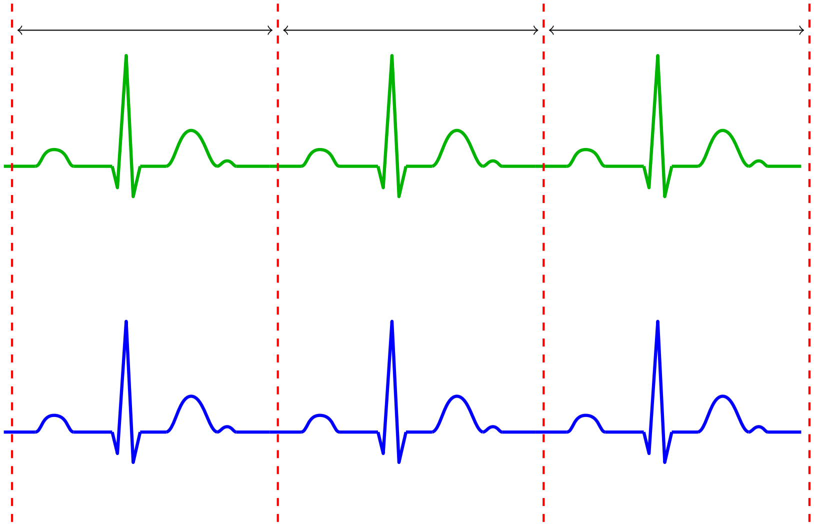


Asynchronous model

No timing assumption about processes and channels,
i.e. no physical assumptions about delays.

- Each process have local view of time called **logical time**
- Any time an event occurs (local or global) at process *p*, its logical clock is updated:
  - local event increase logical time by one unit
  - global event requires more complex strategies (details in a later lecture).

Synchronous model



Synchronous model

Physical timing assumption on processes and links

- *Synchronous computation*: there is known upper bound on processing delays.
- *Synchronous communication*: there is a known upper bound on message transmission delays.
- *Synchronous physical clocks*: Processes are equipped with a local physical clock and there is a known upper bound on the rate at which the local physical clock deviates from a global real time clock

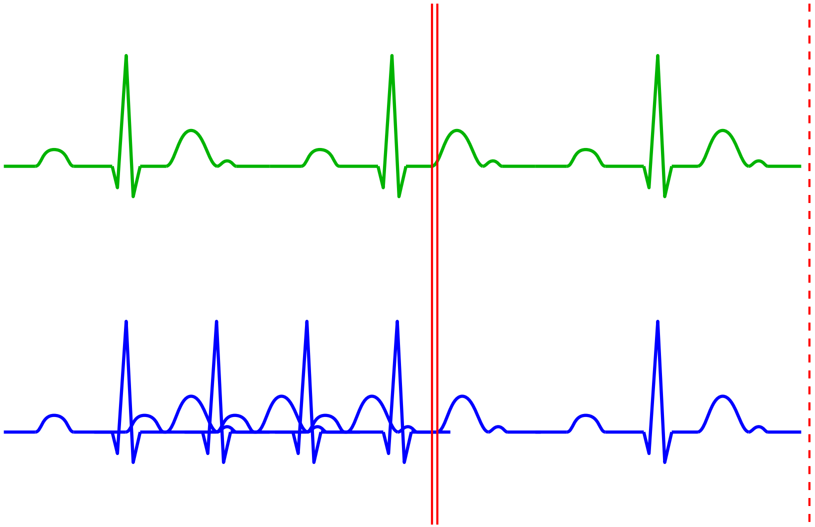
The combination of the computation and communication is called a round

Complexity in Synchronous Model

Two measures of complexity are considered for synchronous distributed algorithms:

- *Time Complexity*: measured in term of number of rounds until all the required outputs are produced.
- *Communication Complexity*: measured in term of non-null messages that are sent. (We may also sometime consider the number of bits in these messages).

Partially Synchronous model



Partially Synchronous model

Most of the time distributed systems are completely synchronous, but there are however periods where the timing assumptions do not hold.

$\Rightarrow$ Assume that a system that is eventually synchronous.

Partially Synchronous model

Most of the time distributed systems are completely synchronous, but there are however periods where the timing assumptions do not hold.

$\Rightarrow$ Assume that a system that is eventually synchronous.

This does **not** mean that:

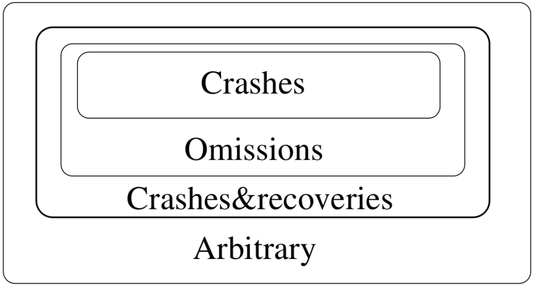
- there is a time after which the underlying system (including application, hardware and networking components) is synchronous forever
- The system needs to be initially asynchronous and then only after some (long time) period becomes synchronous

Table of Contents

Process Failures Model 1/3

Process Failures

Unless it fails, a process is supposed to execute the algorithm assigned to it.



Process Failures Model 2/3

- **Arbitrary fault (Byzantines)**:
    - Not necessarily malicious but can!
    - Can be caused by a bug in the implementation
    - Most expensive to tolerate, but this is the only acceptable option when an extremely high coverage is required
- **Omissions**
    - A process that does not send (resp. receive) a message it is supposed to send (resp. receive), according to its algorithm.
    - Often due to buffer overflows or network congestion
- **Lies**
    - A process that does not send expected responses
    - Often due to malicious behaviour (hacker)

Process Failures Model 3/3

- **Crashes**
  - Special case of omissions where at some point a process do not reply to any message
  - a process executes its algorithm correctly, unless it crashes
- **Recovery**
  - After a crash a process can recovers (a finite number of times)
  - Can be seen as amnesia
  - significantly complicates the design of algorithms because, upon recovery, the process might send new messages that contradict messages that the process might have sent prior to the crash

Link Failure

Crash, Loss, Duplicate can be addressed by some lower level protocol, for instance TCP

As long as the network remains connected, link crashes may be masked by routing.

Link Crashes reveal a lot of impossibility results (see later lectures)

Link Abstraction

- **Fair-loss Links**:
  - Guarantees that a link does not systematically drop any given message
  - Messages might be lost but the probability for a message not to be lost is non-zero
- **Stubborn links**:
  - Make sure its messages are eventually delivered by the destination processes.
  - Keeps on retransmiting all messages sent
- **Perfect links**:
  - Stubborn Links
  - $+$ Duplication avoidance mechanism

Table of Contents

Classical combinations

- **Fail Stop**
  - Perfect links
  - Crash-stop process abstraction
  - Perfect failure detector
- **Fail Noisy**
  - Perfect links
  - Crash-stop process abstraction
  - *Eventual* Perfect failure detector
- **Fail Stop**
  - Perfect links
  - Crash-stop process abstraction
- **Fail Recovery**
  - Stubborn Links
  - Crash-recovery process abstraction

Table of Contents

Basic Properties of Distributed Systems

- **Safety**: states that the algorithm should not do anything wrong
  Example: no process should receive a message unless this message was indeed sent
- **Liveness**: states that eventually something good happens
  Example: if a correct process sends a message to a correct destination process, then the destination process should eventually deliver the message

Temporal Logics

Expressing safety and liveness properties
can be done with temporal logics.
$\Rightarrow$ **Ease the expression of complex properties about multiple distributed processes**

Verification of distributed systems in complex and can't be done with testing
$\Rightarrow$ **Model Checking is a solution (see later lecture)**

Conclusion

- Distributed systems can't be analyzed without abstractions
- The system is modelled as a combination of graph and state machine
- Computing complexity of distributed algorithms is complex, and must consider: classical complexity, message complexity and rounds for synchronous systems
- The system will be analyzed using a combination of link failures and process failures
- The verification of properties of distributed systems fall in two classes : safety and liveness
- This verification can be done with *model checking*