Distributed Systems Models

References:

- Any serious recent distributed systems book 😄

Core elements

- **Computation**: Processes, deterministic vs probabilistic behavior
- **Interaction**: Processes interact through *messages*, which result in:
  - Communication, i.e. information
  - Coordination, i.e. synchronization and ordering of activities
- **Failures**: Which kind of failures can occur?
  - Benign vs malicious (Byzantine)
  - Process vs communication
- **Time**: Determining whether we can make any assumption on time bounds on communication and computation speeds.

Computation

- **Process**: the unit of computation in a distributed system.
  - Sometimes we may call it node, host, etc.
- **Process set**: denoted by Π, it is composed by a collection of *n* uniquely identified processes, like $p_1, p_2, \ldots, p_n$.
- Typical assumptions:
  - The process set is *static* (*n* is well-defined)
    - Processes do know each other
    - All processes run a *copy of the same algorithm*; the sum of all these copies constitutes the *distributed algorithm*
- But in *extreme* distributed systems:
  - The process set is *dynamic* (*n* can vary in time)
  - Too many, too dynamic to know them all
  - Multiple algorithms

Interactions

- Processes communicate through messages
  - *send(m, p)*: sends a message *m* to *p*
  - *receive(m)*: receives a messages *m*
- In some cases, messages may be uniquely identified by
  - Sender of the message
  - A sequence number local to the sender
- General assumption: every pair of processes is connected by a bi-directional **communication channel**
  - Through routing
  - Not true for P2P systems

Process Failures

- In a distributed system, both processes and communication channels may fail, i.e. depart from what is considered its correct behavior.
- **Benign** process failures
  - **Fail-stop**: a process stops executing events, and other processes may detect this fact.
  - **Crash**: a process stops executing events
- **Malicious** process failures
  - **Arbitrary failure** or **Byzantine**: any type of error may occur. This may be caused by:
    - A software bug
    - A malicious behavior inspired by an intelligent adversary

Process Failures

- A process that never fails is **correct**
- A process that eventually fails is **faulty**
- Several distributed algorithms are designed to work correctly if the number of failures f is bounded (e.g., $f < n/3$).

- In some models, processes may perform a **recovery** action:
  - After some time, a process may resume functioning
  - It suffers **amnesia** : the local state maintained in volatile memory is lost
  - To limit the effects of amnesia, a **log** can be maintained

Communication Failures

- **Benign** communication failures
  - Process *p* performs *send* of a message *m* to process *q*
  - Message *m* is inserted in a local outgoing buffer of *p* (**Send-omission**)
  - Message *m* is transmitted from *p* to *q* (**Omission**)
  - Message *m* is inserted in a local incoming buffer of *q* (**Receive-omission**)
  - Process *q* performs receive of *m*
- **Malicious** communication failures
  - Messages created out of nothing, duplicated messages, etc.
  - These problems can easily be solved through *encryption* techniques.

Communication Failures

- Possible causes of message failures:
  - **Buffer overflow** in the operating system
  - **Congestion**
    - Routing errors in routers
  - **Network Partitioning**
    - Processes are subdivided in disjoint sets called *partitions*
    - Communication inside a partition is possible
    - Communication between partitions is not possible
- When a partition disappears, we say that partitions *merge*

Fair Loss Channels

- The channels cannot systematically drop a specific message. This is the minimum abstraction needed to create reliable channels.
- **Fair Loss Channel model**
  - **Validity – Fair Loss**: If a message *m* is sent infinitely often by a process *p* to a process *q* and neither *p* and *q* crash, then *q* will receive *m* infinitely often
  - **Integrity – Finite Duplication**: If a message *m* is sent a finite number of times by a process *p* to a process *q*, then *m* cannot be received by *q* an infinite number of times
  - **Integrity – No creation**: If a message *m* is delivered by some process *p*, then *m* was previously sent by some process *q* to *p*

Correct Channels

- Channels are reliable, messages are never lost. It can be implemented, but there is a price to be payed: *asynchrony*.
- **Perfect Channels model**
  - **Validity – Reliable delivery**: If *p* sends a message to *q*, and neither of *p* and *q* crash, then *q* will *eventually* receive *m*
  - **Integrity – No duplication**: No message is delivered to a process more than once
  - **Integrity – No creation**: If a message *m* is delivered by some process *p*, then m was previously sent by some process *q* to *p*

Time

- **Global clock**
  - For presentation simplicity, it may be convenient to assume the presence of a *global real-time clock*, outside the control of processes.
  - This can be used to provide a *global ordering* of steps in a distributed systems
- **In reality**:
  - Each process is associated with a *local clock*
  - Local clocks may not report the perfect time
  - *Clock drift rate* : refers to the relative amount that a computer clock differs from a perfect reference clock.
- Synchronization is possible, but expensive:
  - Atomic clocks
  - GPS

Asynchronous vs Synchronous

- Distributed systems make difficult to reason about time, not only for lack of clock synchronization. It is also difficult to pose time bounds on events and communication.
- We may think about several different models:
  - **Asynchronous** distributed systems
  - **Synchronous** distributed systems
  - **Partially synchronous** distributed systems

Asynchronous Systems

- There are **no bounds** on the relative **speed of process** execution.
- There are **no bounds** on **message transmission** delays.
- There are **no bounds** on **clock drift**.
  - OR, since we cannot count on their precision at all, **there are no clocks**

- These are not assumptions! These are "*lack of assumptions*"!
- The worst possible model: services as simple as time-based coordination are not possible
- Advantages:
  - simple semantics
  - easier to port to more "powerful" models
  - More realistic: several sources of asynchrony are present in a large-scale network (like the Internet)

Synchronous Systems

- **Synchronous computation**: there is a known upper bound on the relative speed of process execution.
- **Synchronous communication**: there is a known upper bound on message transmission delays.
- **Synchronous clocks**: processes are equipped with local clocks. There is a known upper bound on the drift rates of local clocks with respect to a global real-time clock.

- The best possible model. Can be built, but not with standard hardware/ software.
- Many interesting properties:
  - Timed failure detection (e.g., ping)
  - Coordination based on time (e.g., lease)
  - Worst-case performance analysis
  - Synchronized clocks

Partially Synchronous Systems

- For most systems we know of, it is relatively easy to define physical time bounds that are respected most of the time. There are however periods where the timing assumptions do not hold.
- **Delays on processes**:
  - Machines may run out of memory, slowing down processes
  - A typical case of “no bound on relative speeds of processes”
- **Delays on messages**:
  - Network may congested, and messages may be dropped.
  - Re-transmission protocols can ensure reliability, but at the price of asynchrony
  - Messages may be re-transmitted an arbitrary number of times.