Chapter 1

Basics & System Models

Distributed Systems

SS 2015

Fabian Kuhn

What is a Distributed System?

A distributed system is a collection of individual computing devices that can communicate with each other.

…

Each processor in a distributed system generally has its semiindependent agenda, but for various reasons, including sharing of resources, availability, and fault tolerance, processors need to coordinate their actions.

[Attiya, Welch 2004]

Why are Distributed Systems Important?

Distributed systems are everywhere!

- The Internet
- WWW
- Local area networks, corporate networks, ...
- Parallel architectures, multi-core computers
- Cell phones
- Internet applications
- Peer-to-peer networks
- Data centers
- ...

Why are Distributed Systems Important?

Distributed systems allow to

- share data between different places
- handle much larger amounts of data
- parallelize computations across many machines
- build systems that span large distances
- build communication infrastructures

and also to

- build robust and fault-tolerant systems

Why are Distributed Systems Different?

In distributed systems, we need to deal with many aspects and challenges besides the ones in non-distributed systems.

Some challenges in distributed systems:

- How to organize a distributed system
  - how to share computation / data, communication infrastructure, ...
- There is often no global time
- Coordination of multiple (potentially heterogeneous) nodes
- Agreement on steps to perform
- All of this in the presence of asynchrony (unpredictable delays), message losses, and faulty, lazy, malicious, or selfish nodes

Why Theory?

For distributed systems, we don't have the kind of tools for managing complexity like in standard sequential programming!

Main reason: a lot of inherent **nondeterminism**

- unpredictable delays, failures, actions, concurrency, ...
- no node has a global view
- leads to a lot of **uncertainty**!

It is much harder to get distributed systems right

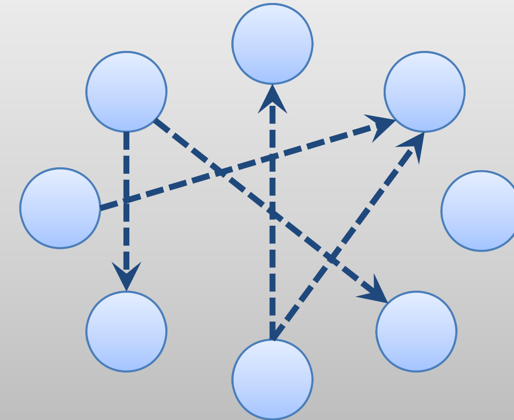
- Important to have theoretical tools to argue about correctness
- Correctness may be theoretical, but an incorrect system has practical impact!
- Easier to go from theory to practice than vice versa ...

Distributed System Models

Two basic abstract models for studying distributed systems...

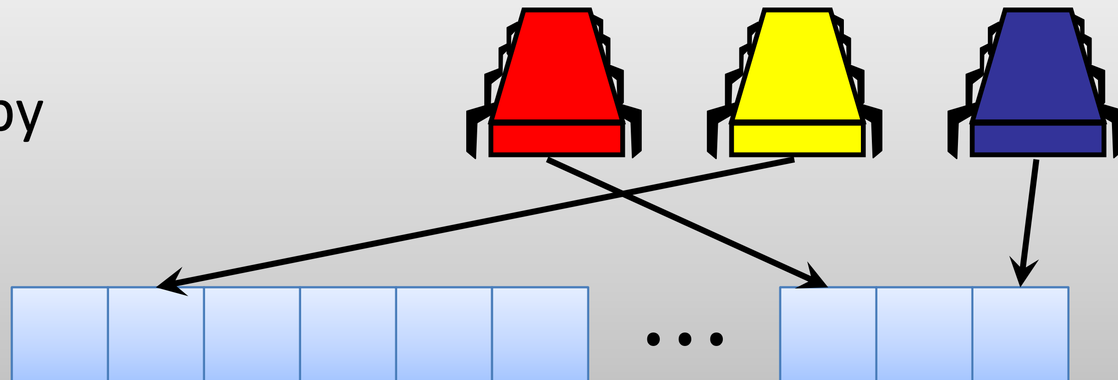
Message Passing:

- Nodes/processes interact by exchanging messages
- Fully connected topology or arbitrary network



Shared Memory:

- Processes interact by reading/writing from/to common global memory



Distributed System Models

Message Passing

- Used to model large (decentralized) systems and networks
- Except for small-scale systems, real systems are implemented based on exchanging messages
- Certainly the right model for large systems that use a large number of machines, but also for many other practical systems

Shared Memory

- Classic model to study many standard coordination problems
- Models multi-core processors and also multi-threaded programs on a single machine
- Most convenient abstraction for programming

Distributed System Models

Message Passing vs. Shared Memory

- Generally, the two models can simulate each other
  - One can implement the functionality of a shared memory system based on exchanging messages
  - One can implement the functionality of a message passing system based on using a shared memory
- Most things we discuss hold for both models
- We will study both models and we will switch back and forth between the models (as convenient)

Synchrony

Synchronous systems:

- System runs in synchronous time steps (usually called **rounds**)
  - Discrete time $0, 1, 2, 3, 4, \ldots$
  - Round r takes place between time $r - 1$ and time r

Synchronous message passing:

- **Round r:**
  At time $r - 1$, each process sends out messages (or a single msg.)
  Messages are delivered and processed at time r

Synchronous shared memory:

- In each round (at each time step), every process can access one memory cell

Synchrony

Asynchronous systems:

- Process speeds and message delays are finite but otherwise completely unpredictable
- Assumption: process speeds / message delays are determined in a worst-case way by an adversarial scheduler

Asynchronous message passing:

- Messages are always delivered (in failure-free executions)
- Message delays are arbitrary (chosen by an adversary)

Asynchronous shared memory:

- All processes eventually do their next steps (if failure-free)
- Process speeds are arbitrary (chosen by an adversary)

Synchrony

There are modeling assumptions between completely synchronous and completely asynchronous systems.

- **Bounded message delays / process speeds:**
  Nodes can measure time differences and there is a (known) upper bound T on message delays / time to perform 1 step.
  - Model is **equivalent to the synchronous model**
  - 1 round $= T$ time units

- **Partial synchrony:**
  There is an upper bound on message delays / process speeds
  - Variant 1: upper bound is not known to the nodes / processes
  - Variant 2: upper bound only starts to hold at some unknown time

Failures

Crash Failure:

- A node / process stops working at some point in the execution
- Can be in the middle of a round (in synchronous systems)
  - some of the messages might already be transmitted...

Byzantine Failure:

- A node / process (starts) behaving in a completely arbitrary way
- Different Byzantine nodes might collude

Omission Failure:

- Node / process / communication link stops working temporarily
- E.g., some messages get lost

Resilience:

- Number of failing nodes / processes tolerated

Correctness of Distributed Systems

When dealing with distributed systems and protocols, there are different kinds correctness properties.

The three most important ones are...

Safety: Nothing bad every happens

Liveness: Something good eventually happens

Fairness: Something good eventually happens to everyone

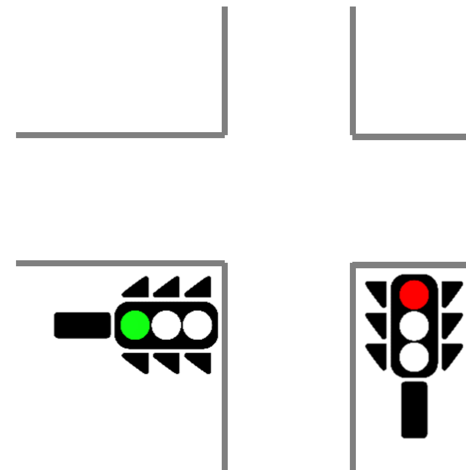
Safety

Nothing bad ever happens.

Equivalent: There are no bad reachable states in the system

Example:

- At each point in time, at most one of the two traffic lights is green.



Proving safety:

- Safety is often proved using invariants
- Every possible state transition keeps a safe system safe

Liveness

UNI FREIBURG

Something good eventually happens.

Example:

- My email is eventually either delivered or returned to me.

Remark:

- Not a property of a system state but of system executions
- Property must start holding at some finite time

Proving liveness:

- Proofs usually depend on other more basic liveness properties, e.g., all messages in the system are eventually delivered

Fairness

Something good eventually happens to everybody.

- Strong kind of liveness property that avoids starvation

Starvation: Some node / process cannot make progress

Example 1: System that provide food to people

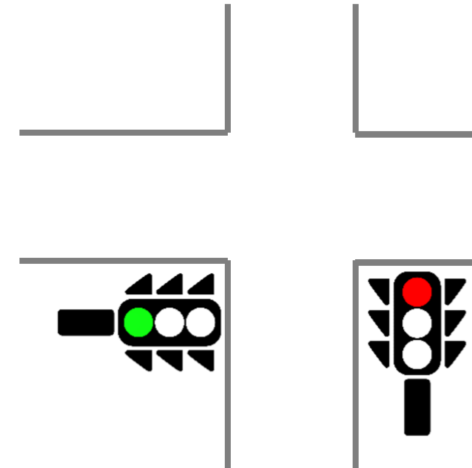
- Liveness properties:
  - Somebody gets food
  - System provides enough food for everybody

Example 2: Mutual Exclusion (exclusive access to some resource)

- Liveness properties:
  - some process can access the resource
  - the resource can be accessed infinitely often

Safety, Liveness and Fairness

Traffic Light Example



Safety: At most one of the two lights is green at each point in time.

Liveness: There is a green light infinitely often

Fairness: Both lights are green infinitely often

Message Passing : More Formally

General remark: We'll try to keep the formalism as low as possible, however some formalism is needed to argue about correctness.

- For detailed models: [Attiya,Welch 2004], [Lynch 1996]

Basic System Model:

1. System consists of n (deterministic) nodes/processes $v_1, \ldots, v_n$ and of pairwise communication channels
   - implicit assumption that nodes are numbered $1, \ldots, n$, n is known
   - sometimes, we want to relax this condition

topology?

2. At each time, each node v_i has some internal state Q_i
3. System is event-based: states change based on discrete events

Event-Based Model

Internal State of a Node:

- Inputs, local variables, possibly some local clocks
- History of the whole sequence of observed events

Types of Events:

- **Send Event:** Some node v_i puts a message on the communication channel to node v_i
- **Receive Event:** Node v_j receives a message
  - must be preceded by a corresponding send event
- **Timing Event:** Event triggered at a node by some local clock

Remarks:

- Events might trigger local computations which might trigger other events

Schedules and Executions

Configuration C: Set (vector) of all n node states (at a given time)

- configuration = system state

Execution Fragment:

Sequence of alternating configurations and events

- Example: $C_0, \phi_1, C_1, \phi_2, C_2, \phi_3, \ldots$
  - C_i are configurations, ϕ_i are events
- Each triple C_{i-1}, ϕ_i, C_i needs to be consistent with the transition rules for event ϕ_i
  - e.g., rcv. event ϕ_i only affects the state of the node that received the msg.

Execution: execution fragment that starts with initial config. C_0

Schedule: execution without the configurations, but including inputs
(the sequence of events of an execution & the inputs)

Message Passing Model: Remarks

Local State:

- State of a node v_i does not include the states of messages sent by v_i (v_i doesn't know if the message has arrived / been lost)

Adversary:

- Within the timing guarantees of the model (synchrony assumptions), execution/schedule is determined in a worst-case way (by an adversary)

Deterministic nodes:

- In the basic model, we assume that nodes are deterministic
- In some cases this will be relaxed and we consider nodes that can flip coins (randomized algorithms)
- Model details / adversary more tricky

Local Schedules

A node v's state is determined by v's inputs and observable events.

Schedule Restriction (seq. of all events) (node v_i)

- Given a schedule S, we define the **restriction $S|i$** as the subsequence of S consisting v_i's inputs and of of all events happening at node v_i

Example:

- 3 nodes v_1, v_2, v_3, send events s_{ij} , receive events r_{ji}
- Schedule $S = s_{13}, s_{23}, s_{31}, r_{13}, s_{32}, r_{31}, r_{23}, s_{13}, s_{21}, r_{31}, r_{12}, r_{32}$

$S|1 =$ $s_{13}, r_{13}, s_{13}, r_{12}$

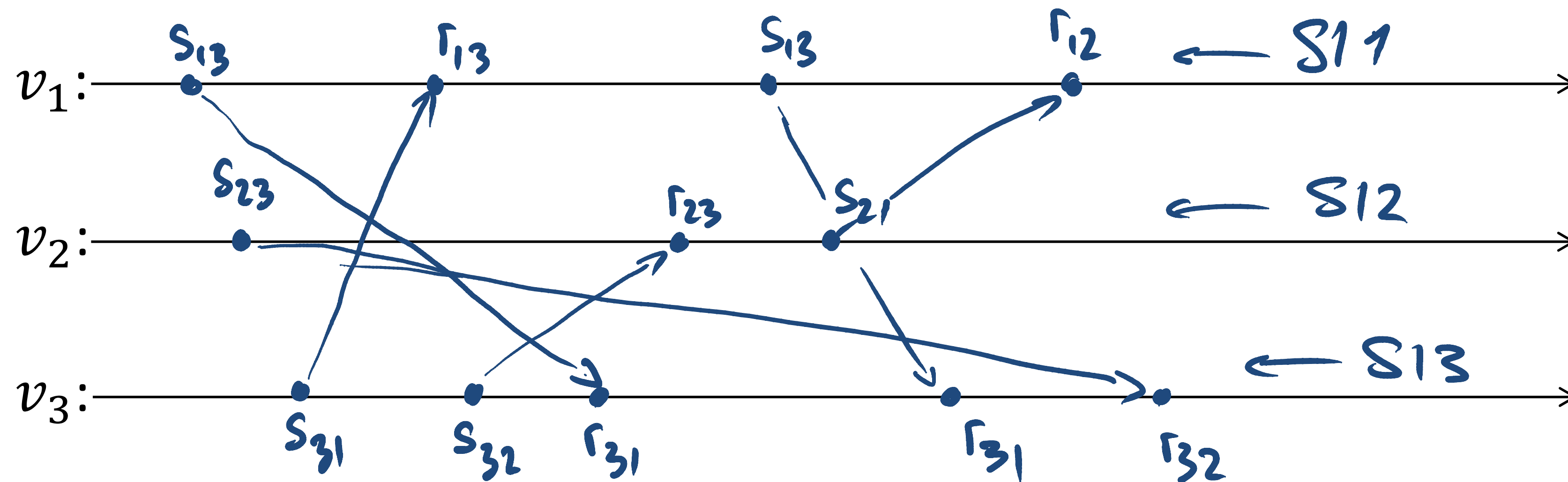
$S|2 =$ $s_{23}, r_{23}, s_{21},$

$S|3 =$ $s_{31}, s_{32}, r_{31}, r_{31}, r_{32}$

Graphical Representation of Executions

Schedule $S = s_{13}, s_{23}, s_{31}, r_{13}, s_{32}, r_{31}, r_{23}, s_{13}, s_{21}, r_{31}, r_{12}, r_{32}$

Graphical representation of schedule / execution



Indistinguishability

Theorem (indistinguishability):
If for two schedules S and S' and for a node v_i with the same inputs in S and S', we have $S|i = S'|i$, if v_i takes the next action, it performs the same action in both schedules S and S'.

Proof:

- State of a node v_i only depends on inputs and on $S|i$
- For deterministic nodes, the next action only depends on the current state.

Lower Bounds / Impossibility Proofs:

- Most lower bounds and impossibility proofs for distributed systems are based on indistinguishability arguments.

Asynchronous Executions

- Only minimal restrictions on when messages are delivered and when local computations are done

A schedule is called **admissible** if

a) there are infinitely many computation steps for each node

b) every message is eventually delivered

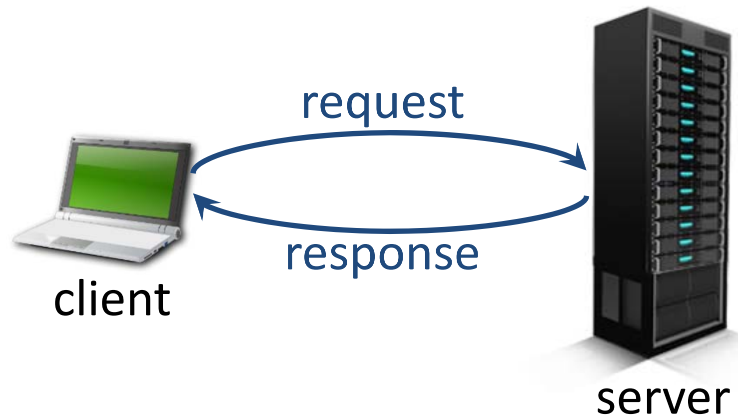
- a) and b) are two fairness conditions

- a) assumes that nodes do not explicitly terminate
- Alternative condition:

  a') every nodes has either infinitely many computation steps or it reaches an explicit halting state

Example: Client-Server Computations

- Most simple kind of interaction, practically extremely important!



Client code:

initially do
  send request to server

upon receiving response do
  process response

Server code:

upon receiving request do
  send response to client

- **Correctness:**
  Schedule is admissible $\Longrightarrow$
  1. After finitely many steps, client sends request
  2. After finitely many steps, request messages is delivered at server
  3. After finitely many steps, server sends response
  4. After finitely many steps, response reaches client
  5. After finitely many steps, client processes response