Programming Distributed Systems

01 Introduction

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2019

reddit
YouTube
amazon
WhatsApp
UBISOFT
TOMTOM
T · · Systems·
Google
IPFS
Akamai
Spotify
bitcoin
PHILIPS
EVERNOTE
Deutsche Bank
NIKE
NETFLIX

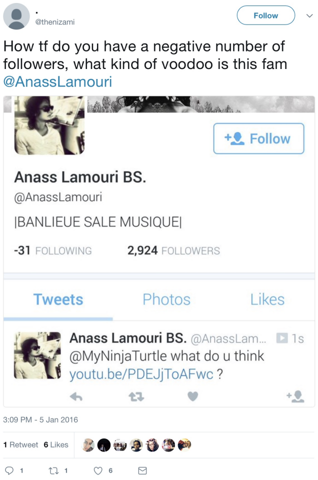
Large-scale distributed systems

All of these applications and systems have something in common:

- Global-scale user base (and users are so annoying with all their demands and expectations)
- Composed of a myriad of services (storage services, web services, membership services, authentication service, . . .)
- Materialized by a huge number of machines, often scattered through-out the world
- Very profitable (with some exceptions . . .)

What can possibly go wrong . . .

Sometimes, voodoo is involved



.
@thenizami

Follow

How tf do you have a negative number of followers, what kind of voodoo is this fam @AnassLamouri

Follow

Anass Lamouri BS.
@AnassLamouri

|BANLIEUE SALE MUSIQUE|

-31 FOLLOWING 2,924 FOLLOWERS

Tweets Photos Likes

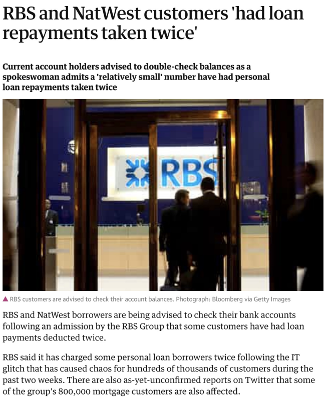
Anass Lamouri BS. @AnassLam... 1s
@MyNinjaTurtle what do u think youtu.be/PDEJjToAFwc ?

3:09 PM - 5 Jan 2016

1 Retweet 6 Likes

1 1 6

Sometimes, problems can be really expensive



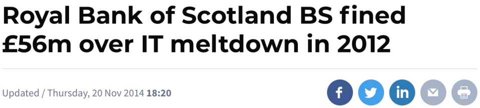
RBS and NatWest customers 'had loan repayments taken twice'

Current account holders advised to double-check balances as a spokeswoman admits a 'relatively small' number have had personal loan repayments taken twice

▲ RBS customers are advised to check their account balances. Photograph: Bloomberg via Getty Images

RBS and NatWest borrowers are being advised to check their bank accounts following an admission by the RBS Group that some customers have had loan payments deducted twice.

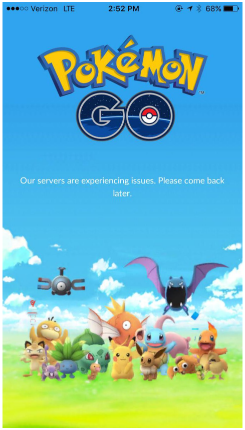
RBS said it has charged some personal loan borrowers twice following the IT glitch that has caused chaos for hundreds of thousands of customers during the past two weeks. There are also as-yet-unconfirmed reports on Twitter that some of the group's 800,000 mortgage customers are also affected.



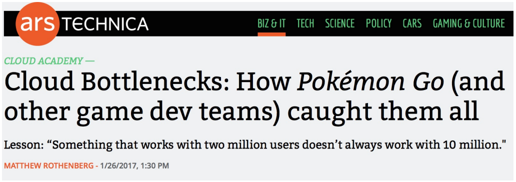
Royal Bank of Scotland BS fined £56m over IT meltdown in 2012

Updated / Thursday, 20 Nov 2014 **18:20**

Sometimes, just everything goes wrong



Our servers are experiencing issues. Please come back later.



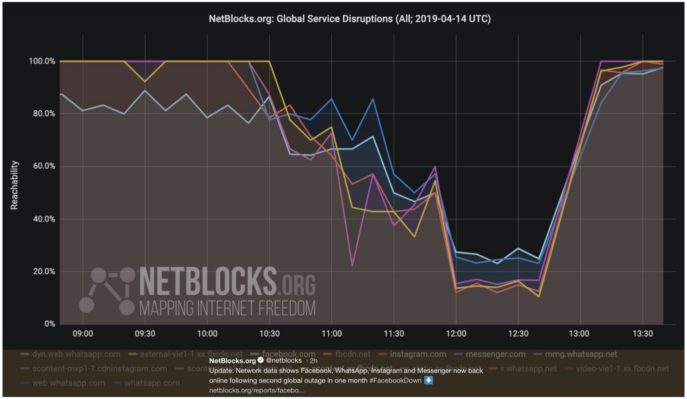
CLOUD ACADEMY —

Cloud Bottlenecks: How *Pokémon Go* (and other game dev teams) caught them all

Lesson: “Something that works with two million users doesn’t always work with 10 million."

MATTHEW ROTHENBERG - 1/26/2017, 1:30 PM

And yesterday. . .



NetBlocks.org @netblocks · 2h
Update: Network data shows Facebook, WhatsApp, Instagram and Messenger now back online following second global outage in one month #FacebookDown ⬇️
netblocks.org/reports/facebo…

The real cost of downtime

For the Fortune 1000, the average total cost of unplanned application downtime per year is $1.25 billion to $2.5 billion.
The average hourly cost of an infrastructure failure is $100,000 per hour.
The average cost of a critical application failure per hour is $500,000 to $1 million.

– Source: Alan Shimal, https://devops.com/real-cost-downtime/, Feb 11, 2015

High availability

Availability %	Downtime per year	per month	per day
90%	36.5 days	72 hours	2.4 hours
95%	18.25 days	36 hours	1.2 hours
99%	3.65 days	7.2 hours	14.4 min
99.5%	1.83 days	3.6 hours	7.2 min
99.9%	8.76 hours	43.8 min	1.44 min
99.99%	52.56 min	4.38 min	8.64 s
99.999%	5.26 min	25.9 s	864.3 ms
99.9999999%	31.5569 ms	2.6297 ms	0.0864 ms

Examples:

- Amazon EC2's: 30% bonus for availability of $<$ 99%/month.
- Google GSuite: Adds 15 days extra for uptime $<$ 95%/month, 3 days for $<$ 99.99%/month.
- Deutsche Telekom: average availability for internet connections is 97%/year.
- Ericsson AXD301, a high-performance highly-reliable ATM switch from 1998, has shown 99.9999999% in 8 month trial period.

Organization of this course

The Basics

- Lecturer: Annette Bieniusa
- Assistant: Peter Zeller

Lectures	Mon + Tue 10:00 - 11:30	Room 48-453
Exercises	Wed 15:30 - 17:00	Room 32-411

Exercises

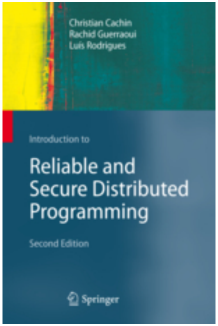
- Mix of theory and practice
  - You will learn a distributed programming language!
  - Implementation of classical algorithms
  - Building a fault-tolerant and resilient middleware
- Bi-weekly exercise sheets
- Final project in second half of term

Checkout installation instructions for Erlang on our webpage!
Bring your laptop on Wednesday!

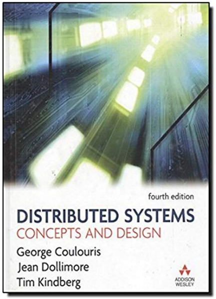
Exam

- Oral exam between August 22-28 or in November
- Registration with examination office (Prüfungsamt) **and** our secretary
- More information later in the course

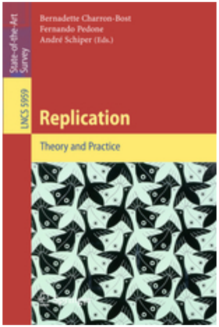
Reading list



[1]



[3]



[2]

Goal of this course

Understanding the intrinsic nature of problems in distributed computing, understanding under which conditions they can be solved, and employing verified and correct modular solutions.

- How do you know what are the components that are currently part of your system?
- How do you propagate information to a large number of nodes (i.e. components)?
- How do you ensure that data is not lost?
- How do you prevent that nodes make inconsistent decisions and mess things up?
- How do you check whether a component (i.e, server) is still active?

Learning objectives

You will be able to

- explain the challenges regarding time and faults in a distributed system
- provide formal definitions for time models, fault models and consistency models
- comprehend and develop models of a distributed system in a process calculus
- describe the algorithms for essential abstractions
- implement basic abstractions for distributed programming
- explain the virtues and limitations of major distributed programming paradigms

Prerequisites

- Very good programming knowledge
- Usage of code repositories
- Basics on network, multi-threading, and synchronization
- Theoretical background (logic, formal languages)

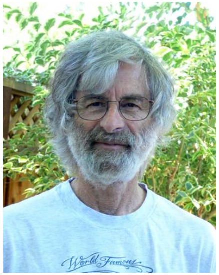
What is a distributed system?

Definition: Distributed system

A distributed system is a model in which components located on networked computers communicate and coordinate their actions by passing messages.

– Coulouris et al. Distributed Systems: Concepts and Design (Addison-Wesley, 2011).

Infamous definition by famous distributed systems researcher



A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
– L. Lamport (ACM Turing Award 2013)

Definition: Service/Server/Client

A **service** is a distinct part of a computer system that mangages a collection of related resources and presents their functionality to users and applications.

A **server** is a running program (i.e. a process) on a networked computer that accepts requests from programms running on other computers to perform a service and respond appropriately.

The requesting processes are **clients**.

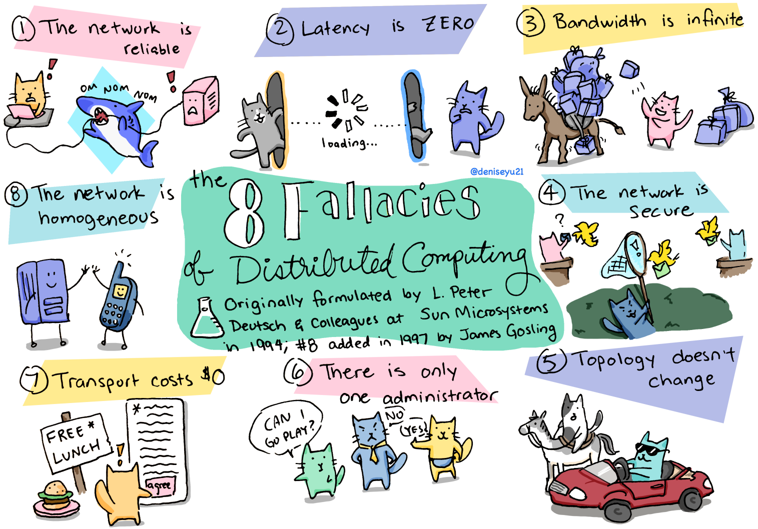
Why do we want to distribute things?

Source: http://www.deniseyu.io/srecon-slides

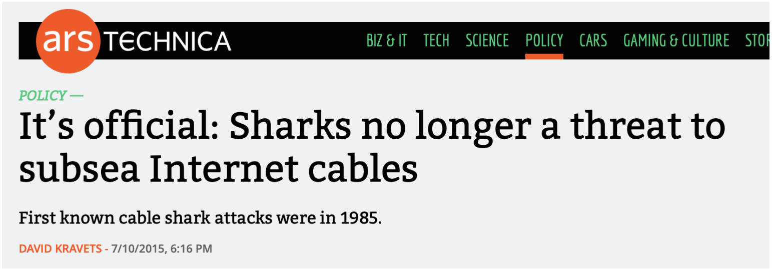
More resources: If, instead of using a single machine to run my system, I use N machines ($N >> 1$), then I will have N times more resources (storage / processing power) and hopefully my system will be (close to) N times faster / answer N times as many requests in the same time unit.

Fault-tolerance (aka dependability): If I use N machines to support my system and f of them ($f < N$) fail, then my system can still operate.

Low latency: A request will be served faster by a machine that is closer to me.


the 8 Fallacies of Distributed Computing
Originally formulated by L. Peter Deutsch & Colleagues at Sun Microsystems in 1994; #8 added in 1997 by James Gosling
@deniseyu21
1 The network is reliable
OM NOM NOM
2 Latency is ZERO
loading...
3 Bandwidth is infinite
4 The network is secure
?
5 Topology doesn't change
6 There is only one administrator
CAN I GO PLAY?
NO
YES!
7 Transport costs $0
FREE* LUNCH
agree
8 The network is homogeneous

Source: http://www.deniseyu.io/srecon-slides



POLICY —

It’s official: Sharks no longer a threat to subsea Internet cables

First known cable shark attacks were in 1985.

DAVID KRAVETS - 7/10/2015, 6:16 PM

Challenges in Distributed Computing

Security

- Confidentiality
- Integrity
- Availability

Scalability

- Handling increase in number of users
- Handling increase in number of resources
- Elasticity

Failure handling

- Detecting failures
- Masking failures
- Tolerating failures
- Recovery

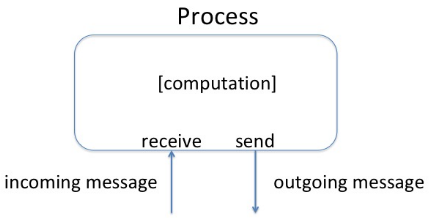
Distributed System Models

Let's go back to the definition

A distributed system is composed by a set of **processes** that are interconnected through some **network** where processes seek to achieve some form of cooperation to execute tasks by sending messages.

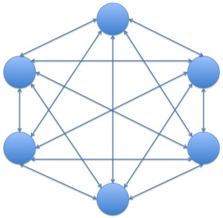
Formal model: Process

- *Processes* are an abstract notion of machine/node.
    - Unless stated otherwise, we assume that all processes of the system run the same local algorithm.
    - Processes communicate through the exchange of messages.
    - Each process is in essence a (deterministic) automaton.



Formal model: Network

- A *network* is modeled as graph $G = (\Pi, E)$ where $\Pi = p_1, \ldots, p_n$ is the set of processes and E represents the communication channels (i.e, links) between pairs of processes.
  - Assumption: Every process is connected to every other by a bidirectional link.
  - In practice: Different topologies can be used, requiring routing algorithms
  - Often, algorithms can be specialized of specific topologies



Assumptions

- A process step consists of *receiving* a message, *executing* a local computation, and *sending* messages to processes.
- Interactions between local components of the same process are viewed as local computation (and not as communication!)
- We can relate a reply message to a response.
  - In practice, this is often achieved by using timestamps based on local clocks.

Time in Distributed Systems

Two fundamental models:

Synchronous System:

We assume that there is a known upper bound to the time required to deliver a message through the network and for a process to make all computations related with the processing of the message.

Asynchronous System:

There are no assumptions about the time required to deliver a message or process a message.

This might look as not a big deal, but actually the timing assumptions have strong implications:

- In a *synchronous* system, you can detect when a process fails (in some particular fault models).
- In a *synchronous* system, you can have protocols evolve in synchronous steps. (Why is that?)
- In an *asynchronous* system, there are some problems that actually cannot be solved.

Synchronous Systems

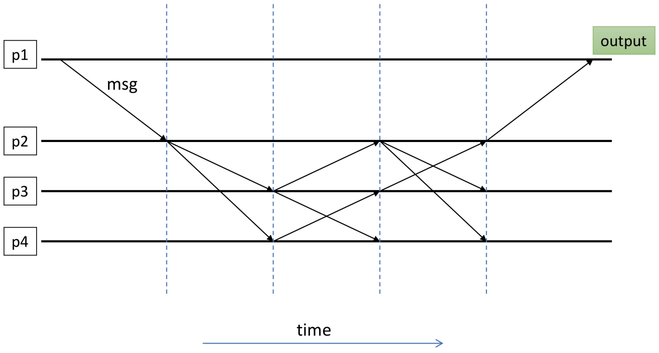
- Known upper bound on computations / message processing.
- Known upper bound on message transmission delays.
- Known upper bound on rate at which local physical clocks deviate from global real-time clock[1]

Example:

- Google's TrueTime API uses atomic clocks, GPS positioning and clever tricks to provide globally synchronized clocks with deviation of less than 6ms.

[1] To simplify the reasoning about the processes, we assume that a global real-time clock exists, but it is not accessible to the processes.

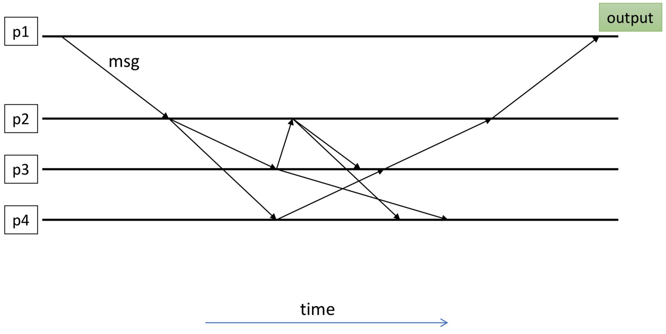
Synchronous Model: Execution in rounds



In each round, a process will:

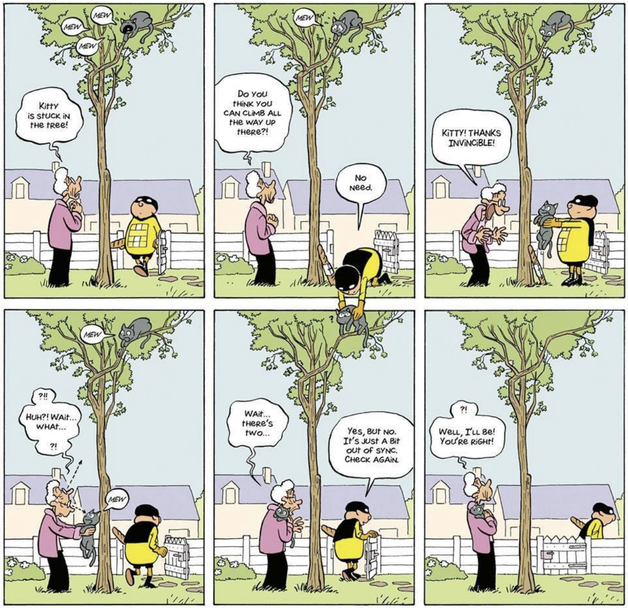
- Receive messages from all processes.
- Process messages to adapt local state and determine which messages are generated.
- Send messages to all processes.

Asynchronous Model: Execution is not based on rounds



Since there is no notion of rounds:

- An (re-)action of a process is triggered by the reception of a single message.
- This can trigger the generation (and transmission) of a new set of messages.


MEW
MEW
MEW
Kitty is stuck in the tree!
MEW
Do you think you can climb all the way up there?!
No need.
KITTY! THANKS INVINCIBLE!
MEW
?!!
HUH?! WAIT... WHAT...
?!
MEW
?
Wait... there's two...
Yes, but no. It's just a bit out of sync. Check again.
?!
Well, I'll be! You're right!

Processes and events

- A system is composed of a collection of **processes**.
- Each process consists of a **sequence** of **events**.

What is an event?

- Depends on concrete model: Can be a single machine instructions or even executing of one procedure
- Sending and receiving of messages are events.

Happens-before Relation

In asynchronous systems, it is only possible to determine a relative order of events[4].

The happens-before relation $\rightarrow$ on the set of events of a system is the smallest relation satisfying the following three conditions:

1. If a and b are events in the same process, and a comes before b, then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Logical clocks

- Each process p keeps a *logical clock* l_p, initially 0.
- When an event that occurs at p is not a receipt of a message, l_p is incremented by 1.
- The value of l_p during the execution (after incrementing l_p) of event e is denoted by $t(e)$ (the timestamp of event e).
- When a process sends a message, it adds a timestamp to the message with value of l_p at time of sending.
- When a process p receives a message m with timestamp l_m, p increments its timestamp to

$$l_p := max(l_p, l_m) + 1$$

We can show:

$$a \to b \quad \Rightarrow \quad t(a) < t(b)$$

Beyond Synchrony and Asynchrony

The “real world” is actually asynchronous, so why is it that we sometimes consider the synchronous model?

Beyond Synchrony and Asynchrony

The “real world” is actually asynchronous, so why is it that we sometimes consider the synchronous model?

- Practical systems are actually *partially synchronous* (or *eventually synchronous*).
- This means that the system is considered to be asynchronous, but it is assumed that eventually (meaning for sure at some time in the future that is unknown) the system will behave in a synchronous way (for long enough).

Fault models

We distinguish between:

- **Fault**: An accidental condition that causes a system component to fail to perform its required function.
- **Error**: An error is a misunderstanding or mistake on the part of a software developer. A fault is introduced into the software as the result of an error.
- **Failure**: Inability of a system component to perform its required function according to its specification.

Example:

Sector in the hard disk is damaged (fault)

$\Rightarrow$ Sector is accessed (error)

$\Rightarrow$ File is lost (failure)

Remarks

- The failure of a component of a process might imply a fault in another (higher-level) component.
- Going back to the previous example, the failure of the file system (file damaged) might lead to a fault in the load of the operative system, which might result in the failure of the operative system.

Process Fault Model

- A process that never fails is *correct*.
- A correct process never deviates from its expected/prescribed behaviour.
- It executes the algorithm as expected and sends all messages prescribed by it.

Remarks:

- Failed processes might deviate from their prescribed behaviour in different ways.
- The unit of failure is the process, i.e., when it fails, all its components fail at the same time.
- The (possible) behaviours of a process that fails is defined by the process fault model.

Classical Fault Models

Crash-Fault Model

- When a process fails, it stops sending any messages (from that point onward).
- This is the fault model that we will consider most of the times.

Omission-Fault Model

- A process that fails omits the transmission (or reception) of any number of messages (e.g. due to buffer overflows).

Fail-Stop Model

- Similar to the crash model, except that upon failure the process “notifies” all other processes of its own failure.

Byzantine (or Arbitrary) Fault Model

- A failed process might deviate from its protocol in **any arbitrary way**.

Examples:

- Duplicate Messages
- Create invalid messages
- Modify values received from other processes

Why is this relevant?

Byzantine (or Arbitrary) Fault Model

- A failed process might deviate from its protocol in **any arbitrary way**.

Examples:

- Duplicate Messages
- Create invalid messages
- Modify values received from other processes

Why is this relevant?

- Can capture memory corruption
- Can capture software bugs
- Can capture a malicious attacker that controls a process

Network Model

The Network Model captures the assumptions made concerning the links that interconnect processes.

Namely, it captures what can go wrong in the network regarding:

- Loss of messages sent between processes
- Possibility of duplication of messages
- Possibility for corruption of messages

Fair-Loss Model

- A model that captures the possibility of messages being lost albeit in a fair way.
- Properties:
  - FL1 (*Fair-Loss*): Considering two correct processes i and j; if i sends a message m to j infinitely often, then j delivers m infinitely often.
  - FL2 (*Finite Duplication*): Considering two correct processes i and j; if i sends a message m to j a finite number of times, then j cannot deliver m infinite times.
  - FL3 (*No Creation*): If a correct process j delivers a message m, then m was sent to j by some process i.

Perfect-Link Model (also called Reliable)

- A stronger model that assumes the links between processes are well behaved.
- Properties:
  - PL1 (*Reliable Delivery*): Considering two correct processes i and j; if i sends a message m to j, then j eventually delivers m.
  - PL2 (*No Duplication*): No message is delivered by a process more than once.
  - PL3 (*No Creation*): If a correct process j delivers a message m, then m was sent to j by some process i.

What about reality?

Our networks are actually closer to the fair-loss model, however its frequent that we use the perfect link model...

Why?

- The perfect link model makes it easier to reason about algorithms design...
- ... but more importantly, these abstractions can be built on top of one another through the use of distributed algorithms.
- In practise:
  - The Fair-loss Point-to-Point Link abstraction can be implemented on UDP sockets.
  - Using TCP sockets, we can implement an abstraction of the Perfect-Link Model.
    - TCP includes acknowledgements and retransmissions
    - Problem in asynchronous systems: Connection is broken if the receiver is unresponsive

Algorithms Specification and Properties

- Notice that when discussing these network models (i.e, abstractions), we have defined them as a set of properties.
- Algorithms (that materialize these abstractions) also provide a set of properties (if correct, those of the abstraction they provide).
- Why do we tend to think in terms of properties?
- Quick answer: Because algorithms are composable, and the design of an algorithm depends on the underlying properties provided by other algorithms.

What does these properties capture?

- The correctness criteria for the algorithm (and its implementation(s))
- It defines restrictions on the valid executions of the algorithm.

Two fundamental types of properties: *Safety & Liveness*

Safety Properties

- Conditions that must be enforced at any point of the execution
- Intuitively, bad things that should never happen.
- Relevant aspects:
  - The trace of an empty execution is always safe (do nothing and you shall do nothing wrong).
  - The prefix of a trace that does not violate safety, will never violate safety.

Liveness Properties

- Conditions that should be enforced at some point of an execution
- Intuitively, good things that should happen eventually.
- Relevant aspects:
  - One can always extend the trace of an execution in a way that will respect liveness conditions (if you haven't done anything good yet, you might do it next).

Safety vs Liveness Properties

Systems are not about lying nor about keeping silent, but about telling the truth!

- Correct algorithms will have both Safety and Liveness properties.
- Some properties however are hard to classify within one of these classes, and they might mix aspects of safety and liveness.
- Usually, one can decompose these properties in simpler ones through conjunctions.

Conclusion: Distributed System Models

A distributed systems model is a combination of

1. a process abstraction,
2. a link abstraction, and
3. a timing abstraction.

Our default model: Fail-stop model

- Crash-stop process abstraction (no recovery)
- Perfect Point-to-Point links
- Asynchronous, but assuming that we can detect crashed processes

Next lecture: The Broadcast Problem

Informally: A process needs to transmit the same message m to N other processes.

Assumptions:

- Complete set of processes in the system is known a-priori
- Perfect Link Abstraction
- Asynchronous system (no rounds, no failure detection)

Further reading I

[1] Christian Cachin, Rachid Guerraoui und Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011. ISBN: 978-3-642-15259-7. DOI: 10.1007/978-3-642-15260-3. URL: https://doi.org/10.1007/978-3-642-15260-3.

[2] Bernadette Charron-Bost, Fernando Pedone und André Schiper, Hrsg. *Replication: Theory and Practice*. Bd. 5959. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-11293-5. DOI: 10.1007/978-3-642-11294-2. URL: https://doi.org/10.1007/978-3-642-11294-2.

[3] George Coulouris u. a. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011. ISBN: 0132143011, 9780132143011.

Further reading II

[4] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), S. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.