Programming Distributed Systems

01 Foundations

Annette Bieniusa

FB Informatik
TU Kaiserslautern

Summer Term 2020

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

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

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 is a distributed system?

Definition: Distributed system

A distributed system is one in which components located on networked computers communicate and coordinate their actions only by passing messages.[1, p. 2]

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

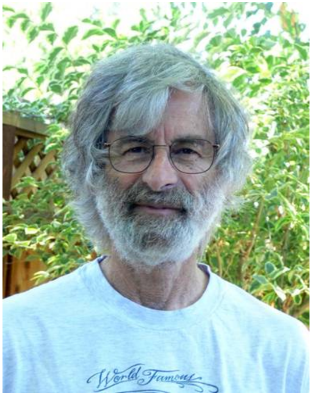
More Definitions [1]

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**.

Infamous definition by famous distributed systems researcher



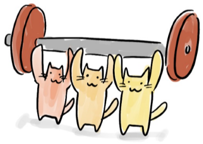
Source: https://commons.wikimedia.org/wiki/File:Leslie_Lamport.jpg

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

Characteristics

- Concurrency of components
- Lack of global time
- Independence of failures

What do we gain by distribution?



From “Why are Distributed Systems so Hard?” by deniseyu is licensed under CC BY 4.0

Scalability: 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 / 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.

Challenges in Distributed Computing I

Security

- Confidentiality
- Integrity
- Availability

Concurrency

- Coordination and Synchronization
- Conflict detection and resolution

Scalability

- Handling increase of requests
- Handling increase of resources
- Elasticity

Challenges in Distributed Computing II

Heterogenity

- Interoperability
- Standardization

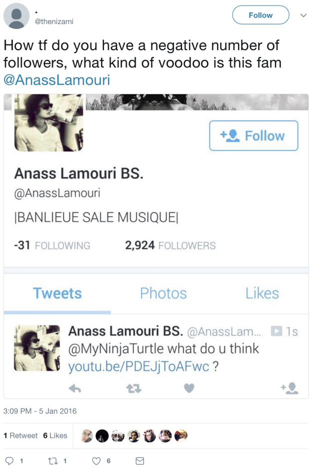
Failure handling

- Detecting failures
- Masking failures
- Tolerating failures
- Recovery

What can possibly go wrong . . .

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

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

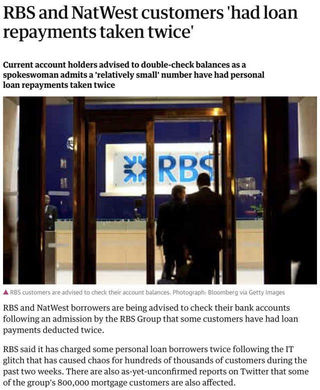
Anass Lamouri BS. @AnassLam... 1s
@MyNinjaTurtle what do u think youtu.be/PDEJjToAFwc ?

3:09 PM - 5 Jan 2016

1 Retweet 6 Likes

1 1 6

Sometimes, problems can be really expensive

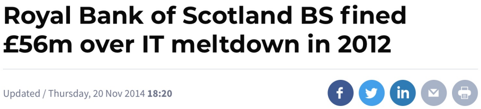

RBS and NatWest customers 'had loan repayments taken twice'

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

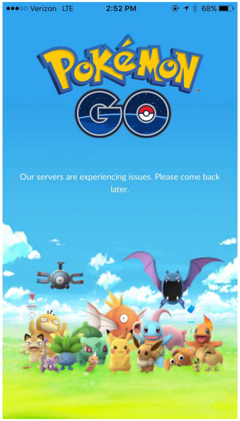
RBS and NatWest borrowers are being advised to check their bank accounts following an admission by the RBS Group that some customers have had loan payments deducted twice.

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.

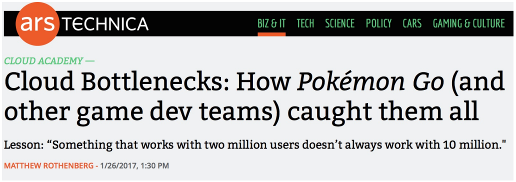

Royal Bank of Scotland BS fined £56m over IT meltdown in 2012

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

Sometimes, everything goes wrong



Our servers are experiencing issues. Please come back later.

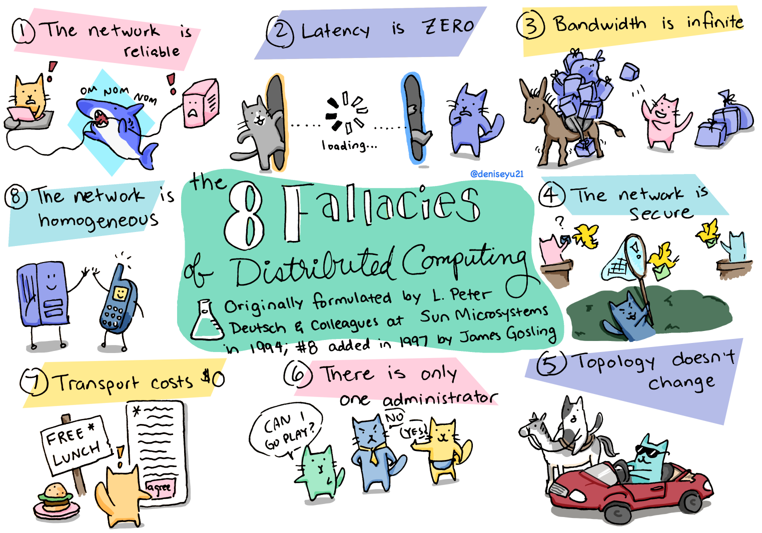


CLOUD ACADEMY —

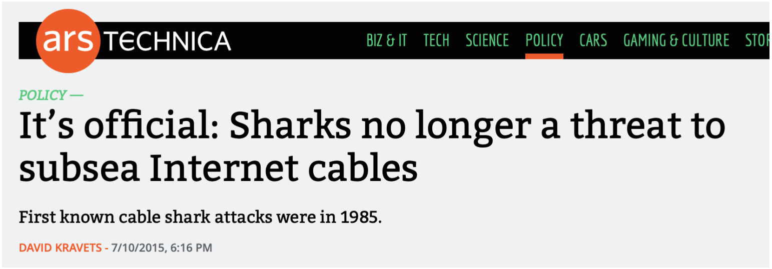
Cloud Bottlenecks: How *Pokémon Go* (and other game dev teams) caught them all

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

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



① The network is reliable
OM NOM NOM
② Latency is ZERO
loading...
③ Bandwidth is infinite
@deniseyu21
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
⑧ The network is homogeneous
④ The network is secure
⑦ Transport costs $0
FREE* LUNCH
agree
⑥ There is only one administrator
CAN I GO PLAY?
NO
YES!
⑤ Topology doesn't change



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

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:

- 1&1 DSL: average availability for internet connections is 97%/year (excluding maintenance).[3]
- Ericsson AXD301, a high-performance highly-reliable ATM switch from 1998, has shown 99.9999999% in 8 month trial period.[2]

Summary

- Major technical developments of the last two decades
- Non-functional requirements define the Quality of Service
  - Reliability
  - Security
  - Performance
  - Adaptability
- Focus in this course is Scalability and Consistency under Replication

Models for Distributed System

Classification of System Models

1. Physical model
   - What are the types of computing devices and how are they interconnected?
2. Architectural model
   - What are the entities that are communicating?
   - How do they communicate?
   - What are their roles and responsibilities?
   - What is their placement in the physical infrastructure?
3. Fundamental model
   - What are the relevant aspects of a system? $\Rightarrow$ Simplifications of complex real-world systems
   - How can we generalize specific problems or findings? $\Rightarrow$ Distributed algorithms and impossibility results

Fundamental model

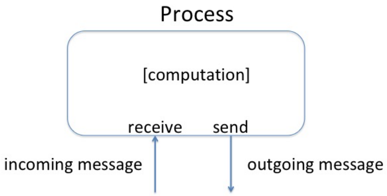
1. Interaction model
2. Failure model
3. Security model

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. [1, p. 2]

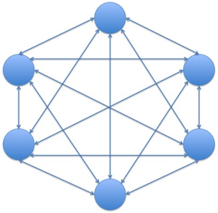
Formal model: Process

- *Processes* $\Pi = p_1, \ldots, p_n$ 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 Π 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 for 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 identifiers or timestamps based on local clocks.

Interaction model

Communication

- Latency
- Bandwidth
- Jitter

Coordination

- Clocks and timing events for ordering of events
- Clock-drift rate

Time in Distributed Systems

Synchronous System [4]

- Time to execute each step has known lower and upper bound.
- Messages that have been sent over channels are received within a known bounded time.
- Each process has a local clock with bound on drift from real time.[a]

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

Asynchronous System:

There are no assumptions about the time required to deliver a message, process a message or clock-drift rates.

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.
- In an *asynchronous* system, there are some problems that actually cannot be solved.

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 (e.g. single machine instructions or executing of one procedure)
- Typically, sending and receiving of messages are events.

Happens-before Relation

In asynchronous systems, it is only possible to determine a relative order of events[5].

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 $b \not\rightarrow a$.

Model vs Reality

- Synchronous systems can be built.
  - Google's TrueTime API uses atomic clocks, GPS positioning and clever tricks to provide globally synchronized clocks with deviation of less than 6ms.
- Asynchronous models are realistic in many situations.
- 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

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 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?

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

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 Fault Model

The Network Fault 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:

- **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.
- **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.
- **No Creation**: If a correct process j delivers a message m, then m was sent to j by some process i.

Perfect-Link Model (aka Reliable)

- A stronger model that assumes the links between processes are well behaved.

Properties:

- **Reliable Delivery**: Considering two correct processes i and j; if i sends a message m to j, then j eventually delivers m.
- **No Duplication**: No message is delivered by a process more than once.
- **No Creation**: If a correct process j delivers a message m, then m was sent to j by some process i.

What about reality?

- Fair-loss Point-to-Point Link abstraction $\approx$ message transmission on UDP sockets.

- Perfect-Link Model $\approx$ TCP sockets
  - TCP includes acknowledgements and retransmissions
  - Problem when assuming asynchronous system: Connection is broken if the receiver is unresponsive

Algorithms Specification and Properties

- 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 do these properties capture?

- Correctness criteria for the algorithm (and its implementation(s))
- 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”).
  - If every prefix of a trace does not violate safety, the trace 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 such that it 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 are difficult to classify within one of these classes, as they might mix aspects of safety and liveness.
- Usually, one can decompose these properties into simpler ones through conjunctions.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Summary: Models for Distributed Systems

A distributed systems model is a combination of

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

Correct behavior of distributed systems require end-to-end argments and involve checks and mechanisms at many different levels [6].

Further reading I

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

[2] Mats Cronqvist. *The nine nines*. Talk at Erlang Factory SF Bay Area 2010. 2010. URL: https://www.erlang-factory.com/upload/presentations/243/ErlangFactorySFBay2010-MatsCronqvist.pdf.

[3] 1und1 Telecom GmbH. *Allgemeine Geschaeftsbedingungen*. 2020. URL: https://dsl.1und1.de/AgbUebersicht.

[4] Vassos Hadzilacos und Sam Toueg. *A Modular Approach to Fault-Tolerant Broadcasts and Related Problems*. Techn. Ber. USA, 1994.

[5] 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.

Further reading II

[6] J. H. Saltzer, D. P. Reed und D. D. Clark. “End-to-End Arguments in System Design”. In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), S. 277–288. ISSN: 0734-2071. URL: https://doi.org/10.1145/357401.357402.