Basic Concepts of Distributed Systems

Lecturer: Prof. Dr. Michael Eichberg
Contact: michael.eichberg@dhbw.de

Version: 1.0.1.2

Slides: https://delors.github.io/ds-basic-concepts/folien.en.md.html
https://delors.github.io/ds-basic-concepts/folien.en.md.html.pdf

Reporting errors: https://github.com/Delors/delors.github.io/issues

AI Usage: AI assistants (in particular Claude, but also ChatGPT, Ollama with Gemma/LLama/Qwen, OpenCode with Kimi, etc.) were used to assist in the creation of the slides. This was done, for example, to efficiently generate graphics (i.e. SVG files), or to generate summary tables. Furthermore, AI was used for general quality assurance. Content that may have been suggested by an AI was explicitly validated and adapted whenever it was incorporated.

The following concepts are of central importance for the development of distributed systems and are implemented in many current middleware products.

1. Time in distributed systems

The importance of time in distributed systems

- Updates that are carried out across several systems must be carried out in the correct order.
- Log entries should be made in the correct order.
- Validity of authorizations (e.g. certificates)
- Geographical positioning (e.g. GPS)

Problems when the time is not correct

A recent surge in GPS “spoofing”, a form of digital attack which can send commercial airliners off course, has entered an intriguing new dimension, according to cybersecurity researchers: The ability to hack time. [...]

“We think too much about GPS being a source of position, but it's actually a source of time,” [...] “We're starting to see reports of the clocks on board airplanes during spoofing events start to do weird things." In an interview with Reuters, Munro [at Defcon] cited a recent incident in which an aircraft operated by a major Western airline had its onboard clocks suddenly sent forward by years, causing the plane to lose access to its digitally-encrypted communication systems.

—11. August, 2024 - GPS spoofers 'hack time' on commercial airlines

Real vs. logical time in distributed systems

Logical Time

Logical time allows us to determine a well-defined sequence between events (cf. 🇺🇸 *happened before* relation). This is *often sufficient* for distributed systems.

Real Time

Solar second: refers to the period of time between successive solar equinoxes.

Atomic Second: The reference point is the period of oscillation of a cesium-133 atom.
TAI (Temps Atomique International): Average time of the atomic clocks of over 60 institutes worldwide (e.g. Braunschweig), determined by the BIH (Bureau International de l'Heure) in Paris.

UTC (Universal Coordinated Time):
Based on TAI; it is currently still necessary to insert occasional leap seconds to adjust to the solar day. The leap second is expected to be abolished from 2035.

Computer Clock Time

- Real-time Clock (RTC): internal battery-buffered clock.
  (The accuracy and resolution are sometimes very coarse).
- Radio-controlled clock (DCF77 from Mainflingen, approx. 2000 km range)
- GPS signal (Global Positioning System) with a resolution of approx. 100 ns
- by means of message exchange with a time server

Clock synchronization according to Christian

(Probabilistic Clock Synchronization, 1989)

- Prerequisite: central time server with UTC.
- Clients ask periodically and correct by half the response time
- Client clocks are never reset but only slowed down or accelerated if necessary.

Network Time Protocol (NTP, RFC 5905)

- Synchronization to UTC
  - in the local network with an accuracy of up to 200 microseconds
  - on the Internet with an accuracy of 1-10 milliseconds
- Hierarchy of time servers

  Stratum 0: Source - e.g. DCF77 time signal transmitter
  Stratum 1: Primary server
  Stratum 2,...: Secondary-/...server
  Clients
- Mutual exchange of time stamps between the server computers is supported (NTP is symmetrical).

However, the time of an NTP server is only updated if the requesting server has a higher *stratum*value (i.e. is potentially less precise) than the requested server. The requesting server then receives the stratum value of the queried server $+1$.

Time: Calculation of the round trip time and the time difference

Origin T_1	System time of the client when sending the request
Receive T_2	System time of the server when the request is received
Transmit T_3	System time of the server when sending the response
Destination T_4	System time of the client when receiving the response

$$\text{RTT:} \quad r = (T_4 - T_1) - (T_3 - T_2)$$

$$\text{Time difference:} \quad x = \frac{(T_2 - T_1) - (T_4 - T_3)}{2}$$

▲ Attention!

Exact clock synchronization cannot be achieved in an asynchronous system!

It is assumed that time passes at virtually the same speed on both computers. The time difference between the two computers is therefore constant.

$(T_3 - T_2)$ is the time required by the server for processing.

The round trip time (RTT) is the time it takes for a signal to travel from one computer to another and back.

The time difference is the difference between the time on the server and the time on the client.

Problems with clock synchronization arise due to uncertain latencies:

- Message transmission time (depending on distance and medium)
- Time delay in routers during relaying (load-dependent)
- Time until interrupt acceptance in the operating system (context-dependent)
- Time for copying buffers (load-dependent)

Due to these problems, a consistent, realistic global snapshot cannot be realized.

Example of Calculating the Time Difference

Let the latency be 5 ms and the processing time 2 ms.

Furthermore, let $T_1 = 110$ and $T_2 = 100$. I.e. the client is ahead.
Since the processing time of the server is 2 ms, the following applies for T_3 and T_4:

$T_3 = 102$ and

$T_4 = 110 + (2 \times 5) + 2 = 122$.
This results in the time difference:
$x = \frac{(100-110)-(122-102)}{2} = \frac{(-10-20)}{2} = -15$ ms.

Logical Time

Observation

For the consistent view of events in a distributed system, the real time is not important in many cases!

We only need a globally unique sequence of events, i.e. we need timestamps.

However, not all events influence each other, i.e. they are causally independent.

It is important to know what happened before and what happened after, but it is not important that we know exactly when (time) something happened.

Lamport Clocks (*logical clocks*)

> Definition
>
> An event (*write*, *send*, *receive*) is a change in a process.

Procedure

- before *write* and *send*: increment the local time $T_{local} = T_{local} + 1$
- *send* always include the timestamp: $T_{msg} = T_{local}$
- before *receive*: $T_{local} = max(T_{msg}, T_{local}) + 1$

The *receive* event is always after *send*.

Events are categorized according to the happened-before relation: **a** → **b**

(a happened-before b)
The result is a partial ordering of the events.

A consistent snapshot contains the corresponding send event for each receive event.

Lamport clocks are one way of supporting *totally-ordered multicasts*, which is particularly necessary in combination with replication.

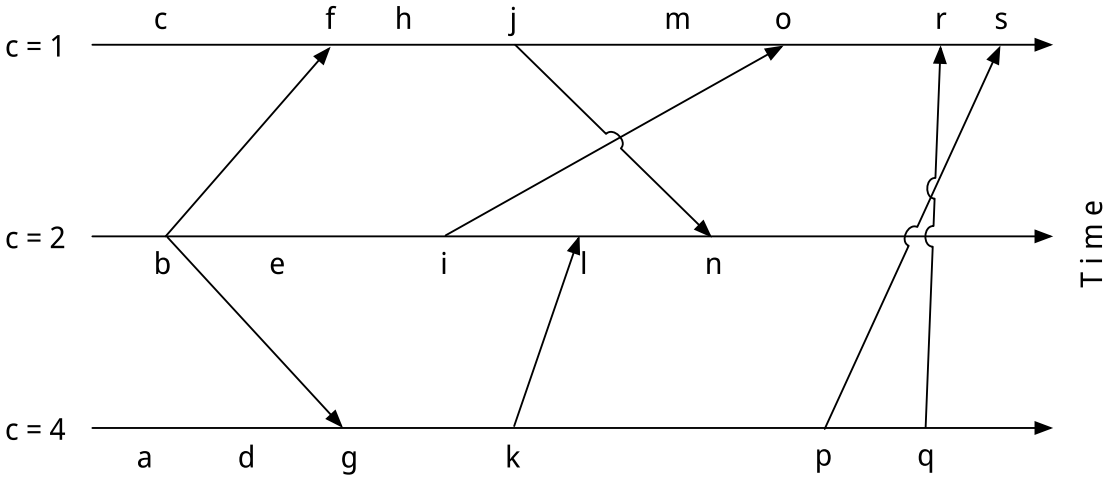
✎ Exercise

1.1. Lamport-Clocks

Consider the following situation with three processes in a distributed system. The timestamps of the events are assigned using Lamport's clocks. (The values c on the far left indicate the state of the respective clocks at the beginning).

a. Provide all events with the correct timestamps.

b. Specify a consistent save point containing event r.



2. Distributed transactions

„Atomic Commit Protocol“

- Distributed transactions extend over several processes and usually also over several nodes in a distributed system.
- More error cases must be taken into account.

  One example would be the transfer of a sum of money (conceptual example):

```
1 send_money(A, B, amount) {
2   Begin_Transaction();
3   if (A.balance - amount ⩾ 0) {
4     A.balance = A.balance - amount;
5     B.balance = B.balance + amount;
6     Commit_Transaction();
7   } else {
8     Abort_Transaction();
9 } }
```

We need an *Atomic Commit Protocol*.

Repetition: Transactions

A transaction ensures the reliable processing of persistent data - even in error situations. The central feature is the guarantee of the ACID properties (Atomicity, Consistency, Isolation, Durability).

At the end of a transaction, either a commit or abort / rollback takes place.

After a commit, all changes are permanent.

Fault tolerance

The aim is to make it possible to build a reliable system from unreliable components.

Three basic steps:

1. error detection: recognizing the presence of an error in a data value or a control signal
2. fault localization: limiting the propagation of faults
3. masking errors: developing mechanisms that ensure that a system functions correctly despite an error (and possibly corrects an error)

Two-Phase Commit Protocol - 2PC

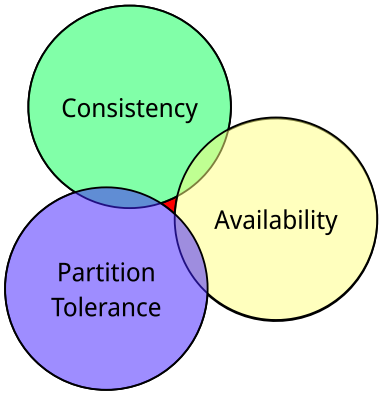
Participants are (1) those (P_i), who manage the distributed data, and (2) a coordinator (K), who controls the protocol. (K may itself be one of the P_i)

1. **Coordination Phase**:
   - K sends a PREPARE message to all P_i.
   - Each P_i checks for itself whether the transaction can be completed correctly locally.
   - If so, it sends READY, otherwise ABORT to K
2. **Decision Phase**:
   - If all P_i have responded with READY, K sends COMMIT to all P_i; otherwise K sends an ABORT message to all P_i
   - If the decision was COMMIT, all P_i make the transaction *stable*
   - If the decision was ABORT, all P_i roll back the transaction.
   - Finally, all P_i send an OK message to K

The 2-PC protocol is not error-resistant, i.e. it can recognize errors but cannot necessarily correct them. To handle some error scenarios, results (especially READY and COMMIT) must be recorded in a persistent *write-ahead* log file.

CAP Theorem[1]

In **distributed** (*database*)*systems*, only two of the following three properties can be guaranteed at the same time:



- *Consistency*

  After completion of a transaction, the return value of the next read operation is the result of the last write operation or an error.

- *Availability*

  Each request receives a response in an acceptable time.

- *Partition tolerance*

  The system also works with network partitioning, i.e. nodes can no longer communicate with each other.

The CAP theorem *only* refers to distributed systems. Network partitioning can always occur in such systems. Therefore, partition tolerance is a natural property and you can often only choose between consistency and availability.

Which properties are important in which scenarios?

DNS: Availability and partition tolerance

Banking: Consistency and partition tolerance

---

[1] 2000 Brewer(Conjecture), 2002 Gilbert and Lynch(Proof)

✎ Exercise

2.1. Two-Phase-Commit

Analyze how the two-phase commit protocol deals with error situations.

Which errors can occur at which points in time and which can the protocol rectify?