FLEX - Distributed Computing

Simon Härer

Created: 2021-01-19 Tue 08:29

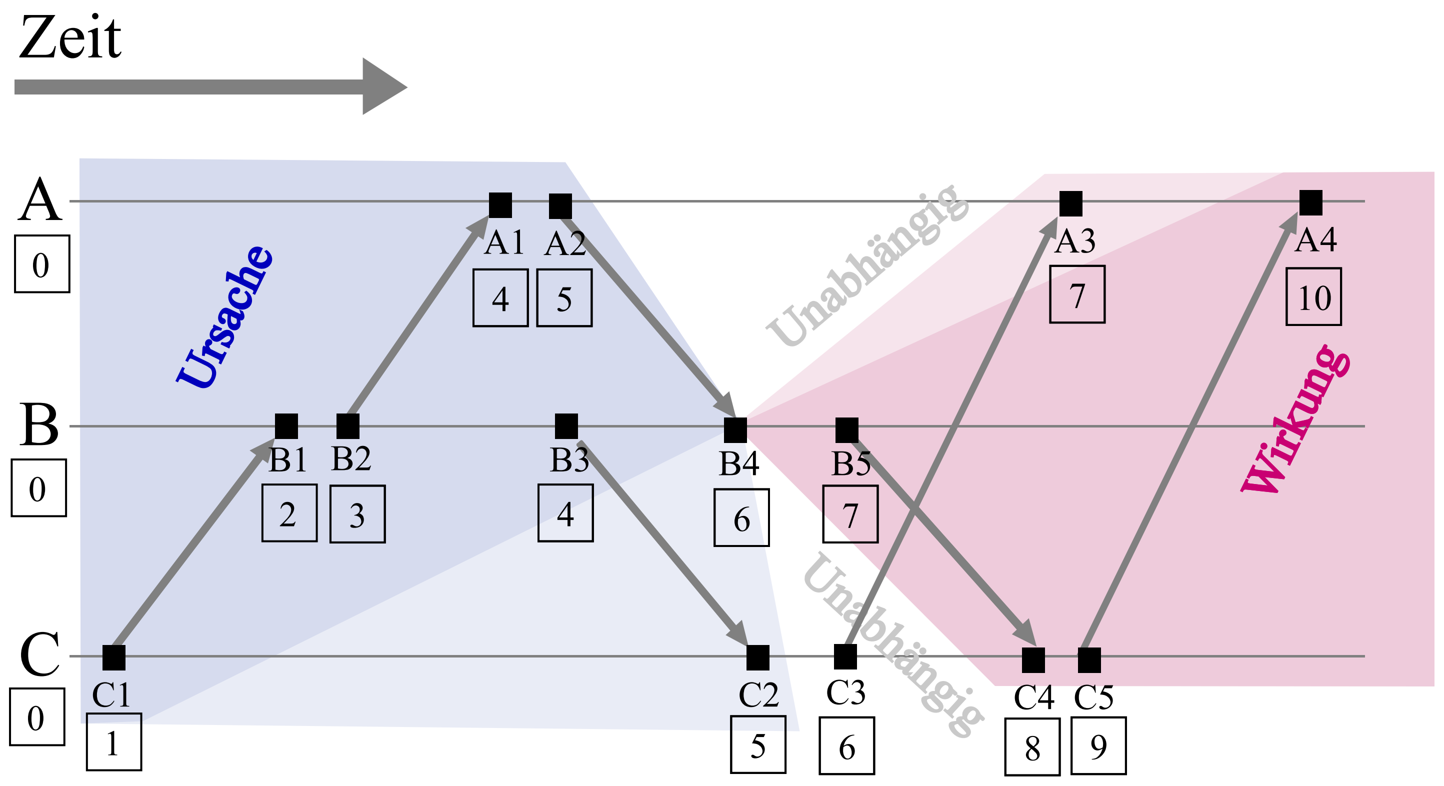
Monolithic Systems

- everything on one machine, in one process
- displayed information is instantly updated
- data storage is synchronously updated

Traditional Stability Approaches

- "pretend monolith"
- synchronous communication
- keep everything running at all costs
- use database transactions to ensure synchronous updates

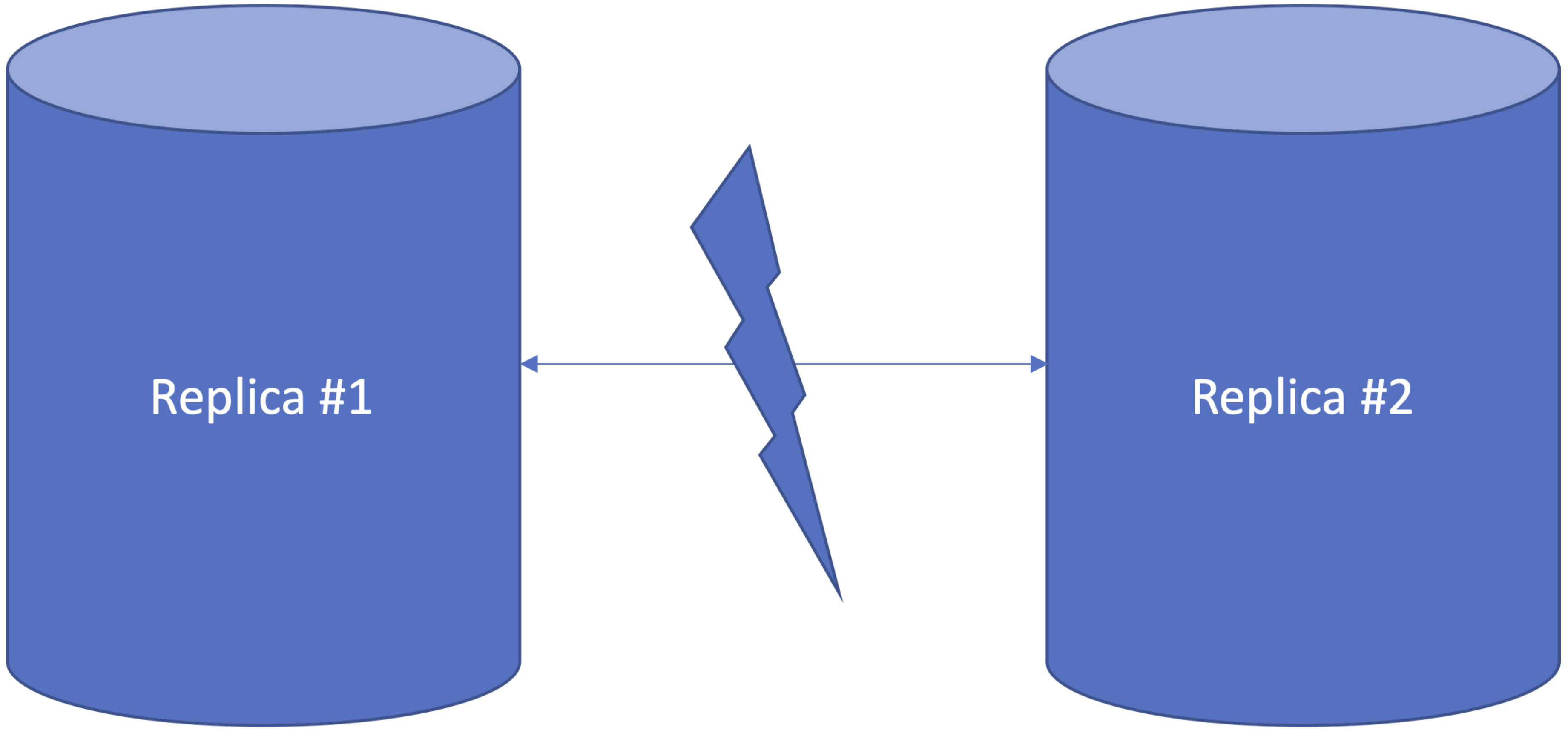
Causality and Event Ordering



Lamport-Uhr (Quelle: Wikipedia)

@ctive group

Split-Brain-Problem



@ctive group

Fallacies of distributed computing

- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn't change
- There is one administrator
- Transport cost is zero

- The network is homogeneous

Rotem-Gal-Oz, Arnon. (2008). *Fallacies of Distributed Computing Explained*. Dr. Dobb's Journal.

The network is reliable

Network failures can and will occur on different levels. Power failures or someone tripping on a network cord are very pragmatic ones. A switch could fail and a package could be lost. When third party networks are included, there is no control of the actual quality at all.

The fallacy leads to software that does little to no network-related error handling. In the likely case of a network error, the application faces unhandled errors. We need to handle that every request can fail and try to limit the risk of failures. This can be achieved by introducing software and hardware redundancy. However, a network always stays unreliable to some certain degree.

Latency is zero

Latency is the time data needs from one place to the other, while bandwidth describes how much data can be moved within a given timeframe. E.g. a simple ping command still needs a considerable amount of time. This gets very relevant if software regularly send requests for example using ajax in a client application.

The fallacy leads to software that doesn't explicitly reason about or manage the frequency of network requests. The software will face timeouts and exhaust bandwidth. We need to make as few requests as possible and transfer as much data as possible each request.

Bandwidth is infinite

While bandwidth improved magnitudes faster than latency, the amount of data of interest equally did. Bandwidth is often beyond our control. Considering the increasing amount of mobile apps, this consideration is more important than ever.

The fallacy leads to software that is very generous with bandwith and will face bottlenecks. Taking care of utilized bandwidth in development naturally imposes a balancing act between it and latency.

The network is secure

Today, software that is publicly available gets attacked by a variety of automated attackers. Even in intranets, once a malicious actor gets inside the network, security measures are important. No network is totally secure.

The fallacy leads to software development, that doesn't even take the avoidance of security flaws into account in the first place. Moreover, it leads to developers that will not regularly apply security patches. When planning software, a threat modelling should be done to evaluate security risk. Security is a key aspect, that should be part of a system's architecture from day one.

Topology doesn't change

Topology doesn't change very much when developing a system. But once deployed, topology will likely change. At this stage, the network topology is often out of our control. Server are added and removed, services are changed and network error lead to changes in topology. When talking about clients, topology is changing constantly.

The fallacy leads to systems that are hard-coded to fit one topology only. With changing topology, it can result in already mentioned errors (latency, bandwidth, security). To be prepared, software should be developed without exact assumptions about topology, e.g. endpoints and routes. Topology can be abstracted and handled by another layer.

There is one administrator

In a very small setup we may get away with having a single administrator. However, as soon as we talk about enterprise grade systems or system that involves third party services, that simple setup is gone.

The fallacy leads to hardly debuggable systems. When other administrators aren't in the same team or even in other companies, communication must be organized before an error occurs. Who is responsible? Did someone change a setting? Who potentially affects the error? Risk of bugs that could originate from multiple scopes should be evaluated and communication processes established.

Transport cost is zero

Neither on application level (e.g. marshalling) nor on network level (actually sending bytes) is this not a fallacy. Serializing data takes time to development, performance at runtime and introduces complexity. Sending data over the network causes costs of bandwith, security measurements, hardware, ...

The fallacy leads to system, where the actual costs get underestimated. This is an economic point to be considered and can be the factor that makes a project profitable or not.

The network is homogeneous

Nowadays, we almost always assume networks to be heterogenous. A lot of standard protocols and exchange formats have been established to handle heterogeneous networks. Take REST or communication based on XML over HTTP as an example. Avoid proprietary protocols or transports.

Exercise: Erlbank – Fallacies of Distributed Computing

Analyze the original Erlbank application with respect to the fallacies.

Answer:

- The network is reliable

  The services communicate asynchronously and use eventual consistency as a consistency model. In case of a service not being reachable, they still function as intended, but won't have other services' current data.

- Latency is zero

  Feed requests can still overtake the former request, we are not safe regarding this. This could mess up the pointer we are maintaining in the state and, thus, make the consumer ignore serveral events. We can improve here, by checking for consecutive event numbers.

Answer:

- Bandwidth is infinite

  On high traffic peaks we could be too slow to fetch new data. A feed-consumer wouldn't be able to catch up and lag behind. It would still function but the consistency delay would increase. We'd need to monitor this and react accordingly.

- The network is secure

  We didn't talk about Erlang's security model yet. There are cookies preventing malicious actors from joining the network. What else do we need to consider?

Answer:

- Topology doesn't change

  Erlang's OTP offers a huge amount of tooling to manage and set-up topology. It helps to change and monitor it.

- There is one administrator

  We will talk about that in the deployment and monitoring chapter.

- Transport cost is zero

  We have neither talked about how Erlang serializes and deserializes data over the network, nor have we considered the impact and performance cost. How would we do that?

Answer:

- The network is homogeneous

  We avoided propiertary transports and protocols. Erlang's OTP utilizes TCP/IP and we are staying withing the framework. We still have a very homogeneous network, since we have all bet on Erlang, but that must not be the case. We are already fetching exchange rates over HTTP.

Resilience

In case of distributed computing, we always have to think about what happens when a service is unreachable from another service. One strategy is to increase uptime another is to program with errors in mind.

There are several patterns that can be used to increase resilience in a distributed setup. We will discuss Retry, Fallback, Circuit breaker and Bulkhead.

Nygard, Michael T.. *Release It!: Design and Deploy Production-Ready Software.* (2017).

Retry Pattern

Very simple:

- A service retries in case a request to another service fails
- Helps in case of e.g. temporary package loss
- Doesn't help if the other service is overloaded, maybe makes it even worse

Fallback

- Define fallback logic if a service is not reachable
- E.g. product ratings cannot be fetched? Just don't show ratings for now
- Not always possible, e.g. what if the product information itself cannot be fetched?

@ctive group

Circuit Breaker

A circuit breaker can have 3 states: Open, closed and half-open.

- In closed state, all requests are handled unconditionally
- In open state, all requests are rejected
- In half-open state, some requests are handled, others rejected

The circuit breaker sits between the requesting service and the responding service.

In case of multiple consecutive (internal) errors in the responding service, the circuit breaker opens. It then rejects incoming requests. After some time, it half-opens and handles some requests. If these are successful again, it resets to close.

Circuit breakers allow controlled error handling and are especially helpful, when a service is overloaded. They work well combined with e.g. the Fallback pattern.

Bulkhead Pattern

Bulkhead is a resource partitioning between many service that normally would access the same resources, e.g. CPU.

- If one service exhaust its resources, it is not reachable anymore
- Due to bulkheads, other services are not affected, they have their own resources
- Helpful combined with Fallback, so that the application is still working in case of one service being overloaded

No Silver Bullet

Patterns and measures depend on:

- requirements
- environment
- people

Availability

Availability = Uptime / (Uptime + Downtime)

Availability = MTBF / (MTBF + MTTR)

MTBF Maximization Strategies

- automated testing before deployment
- "chaos engineering"
- automatic restart of failed components
- reroute to working replicas
- supervisor hierarchies
- event monitoring
- latency monitoring

MTTR Minimization Strategies

- incident-management action plan
- define roles for incident management
- training on roles and functions
- monitor and alert
- anomaly detection
- document
- review incidents

@ctive group

Fault Isolation Strategies

- small components with loose coupling
- expect other components to fail
- horizontal replication
- "let it crash"
- supervisor hierarchies