Mastering the Maze: Practical Strategies for Navigating Complexity in Distributed Systems.

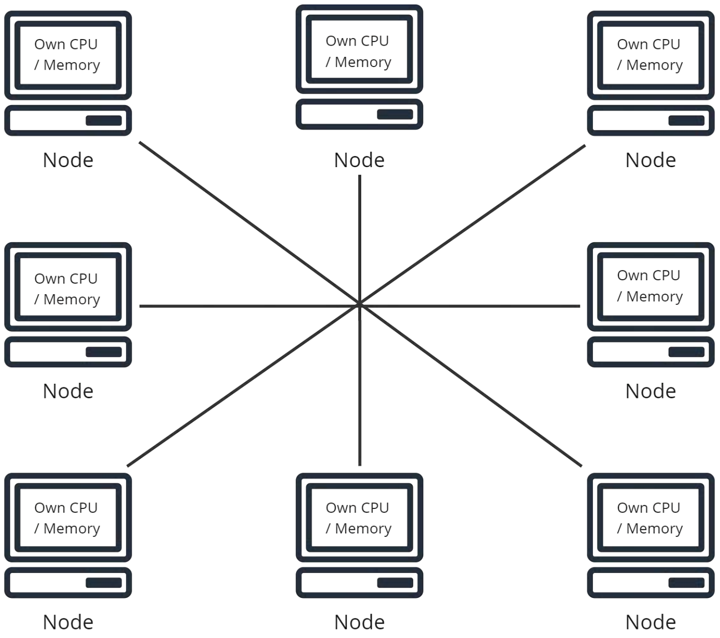
Aleksei Popov, Software Engineering Manager, Stenn International Ltd.

Agenda

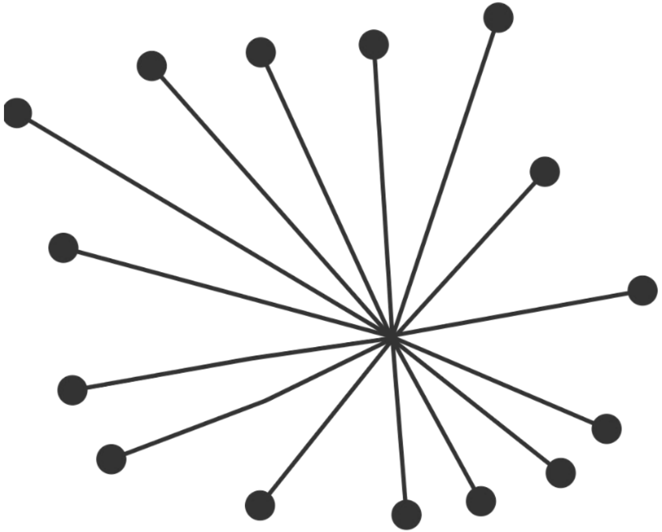
- What is a distributed system?
- What are the main complexities of building distributed systems?
- Core principles of systems engineering and cybernetics
- SRE practices for reducing complexity overhead

What is a distributed system?

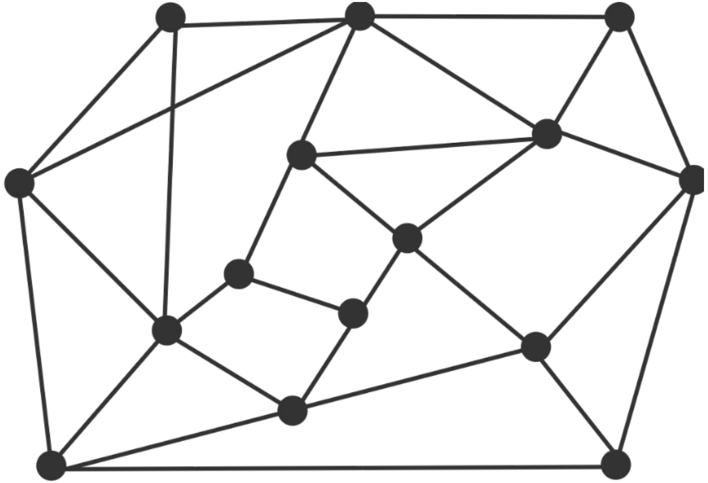
Share nothing



What is a distributed system?



Centralized



Distributed

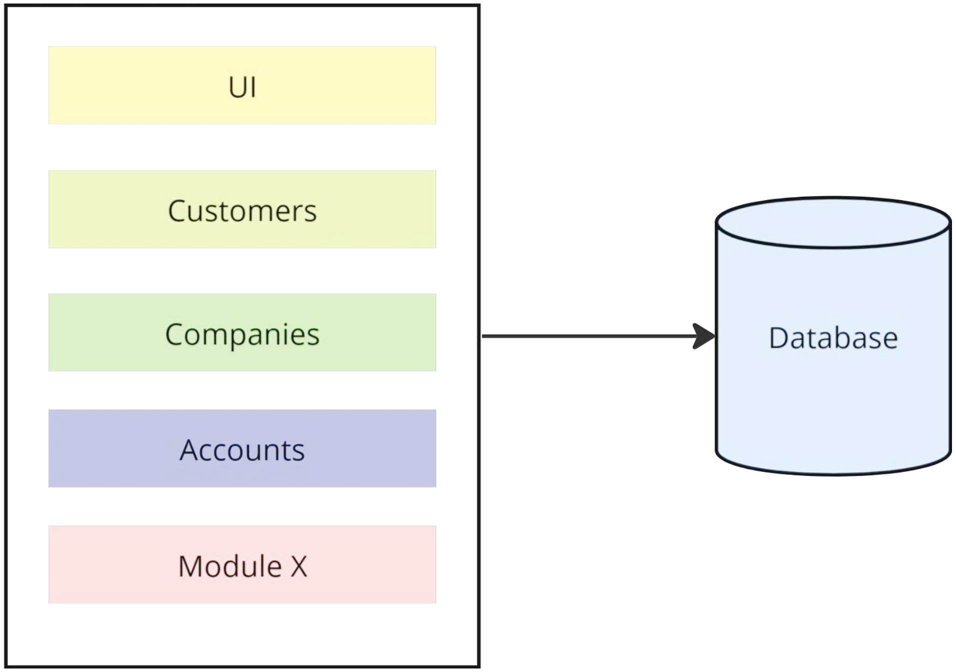
What is complexity?

- In Systems Theory, **complexity** describes how various interdependent parts of a system interact in complex and sometimes surprising ways, resulting in new and unexpected behaviors.

- In Software and Technology, **complexity** refers to the details of software architecture, such as the number of components, how deeply these components depend on each other, and how they interact within the system.

Monolithic Architecture

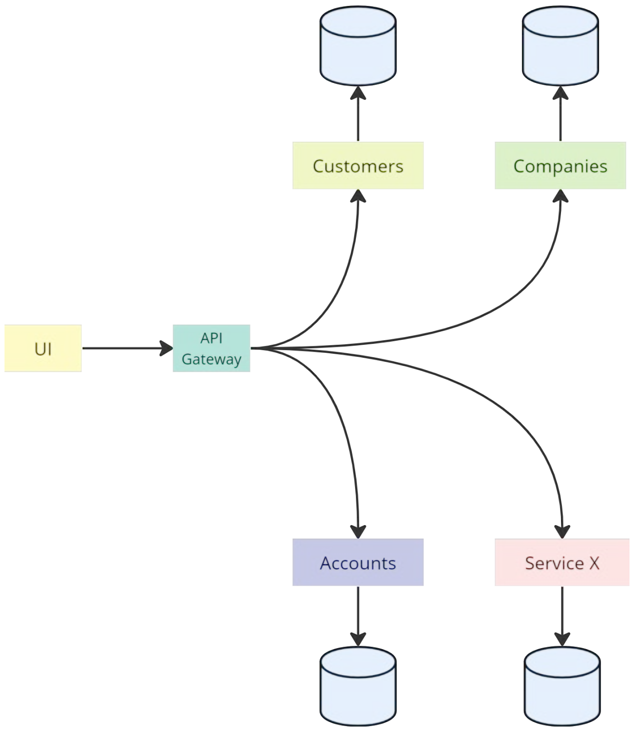
Single Deployable and Executable Component



Monolithic Architecture: Disadvantages

- Inability to scale modules independently
- Harder to control growing complexity
- Lack of modules independent deployment
- Challenging for developers to maintain single huge codebase without well-established rules
- Technology and vendors coupling

Microservices Architecture



A **microservices** architecture implies creation of small and autonomous services which can be implemented, tested and deployed independently.

What do distributed systems give us?

- Horizontal scalability
- High-availability and fault-tolerance
- Geographic Distribution
- Technology choice freedom
- Easier architecture control

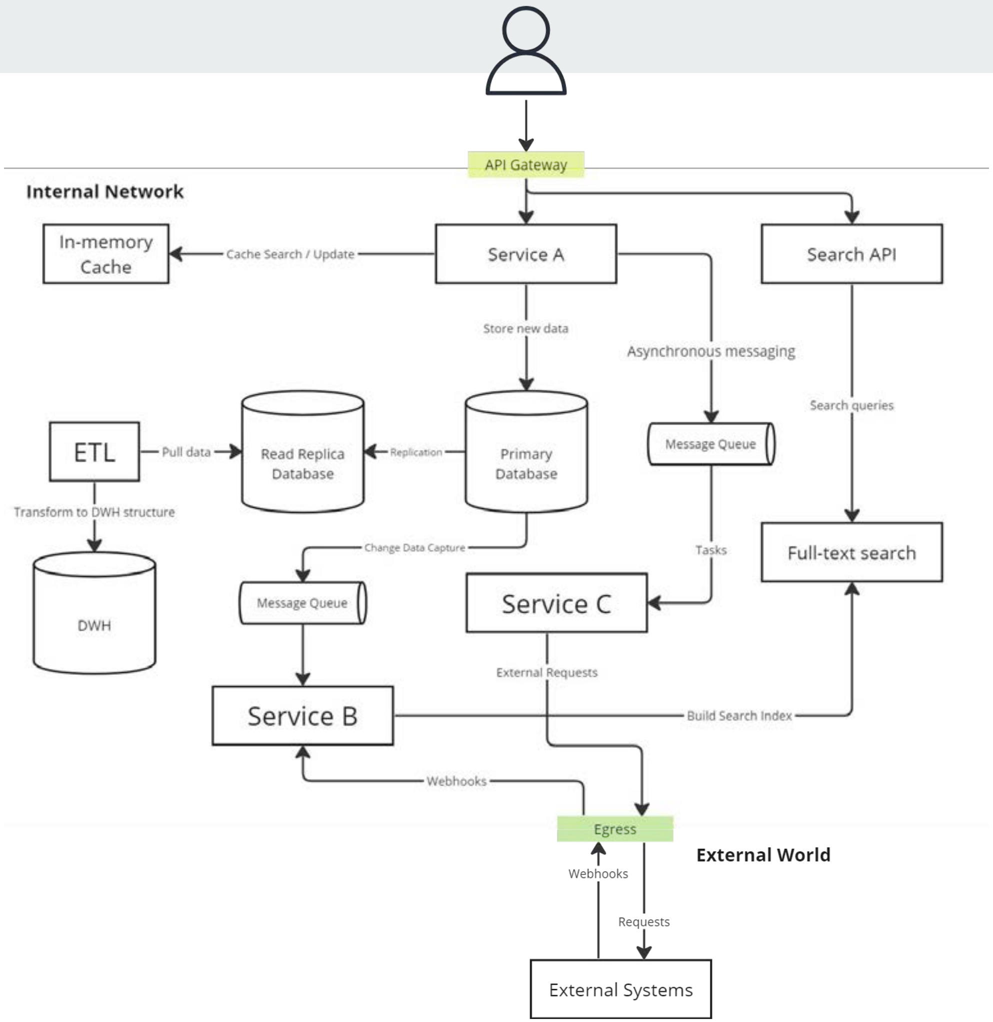
But it also comes with its own challenges

Quality Attributes

Reliability

Scalability

Maintainability


API Gateway
Internal Network
In-memory Cache
Cache Search / Update
Service A
Search API
Store new data
Asynchronous messaging
Search queries
ETL
Pull data
Read Replica Database
Replication
Primary Database
Message Queue
Transform to DWH structure
Change Data Capture
Tasks
Full-text search
DWH
Message Queue
Service C
External Requests
Service B
Build Search Index
Webhooks
Egress
External World
Webhooks
Requests
External Systems

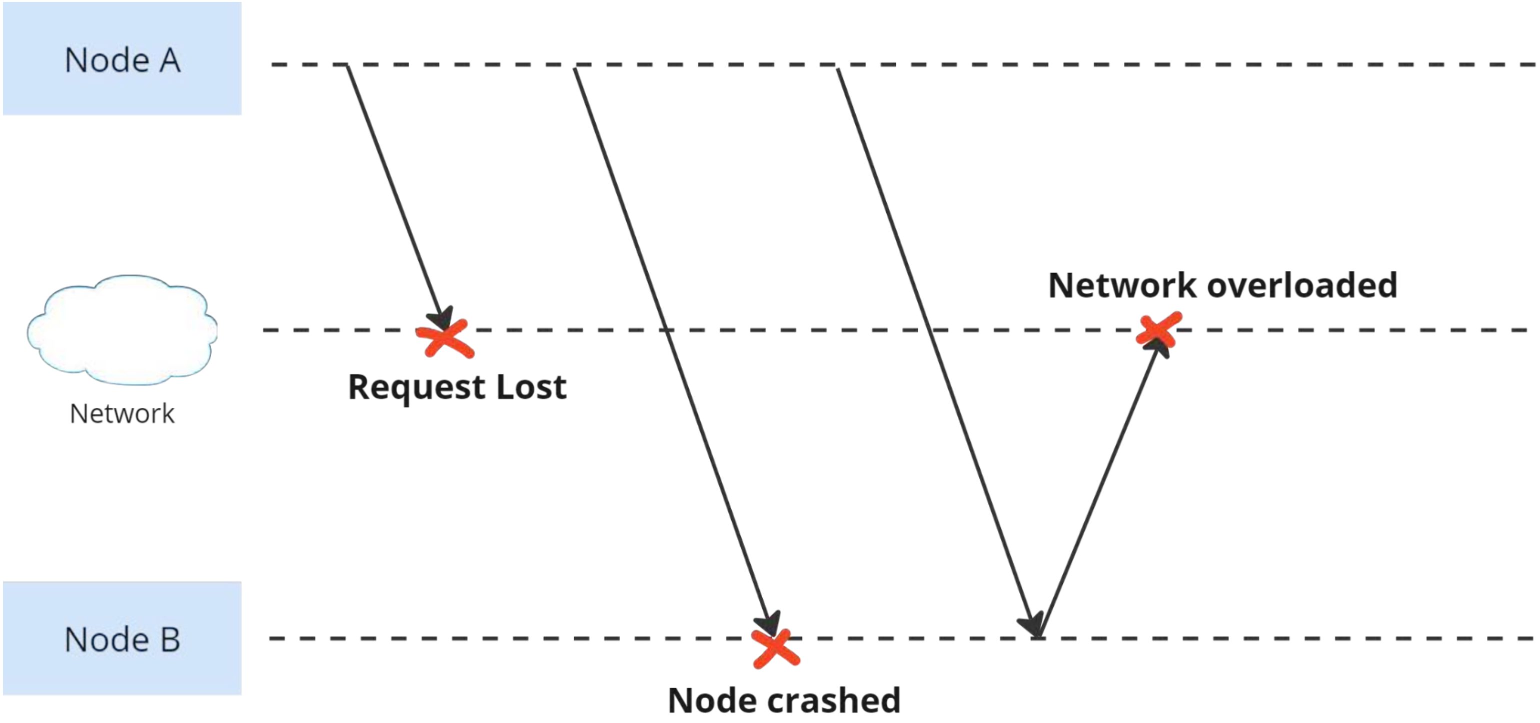
“Anything That Can Go Wrong, Will Go Wrong”

Murphy's law

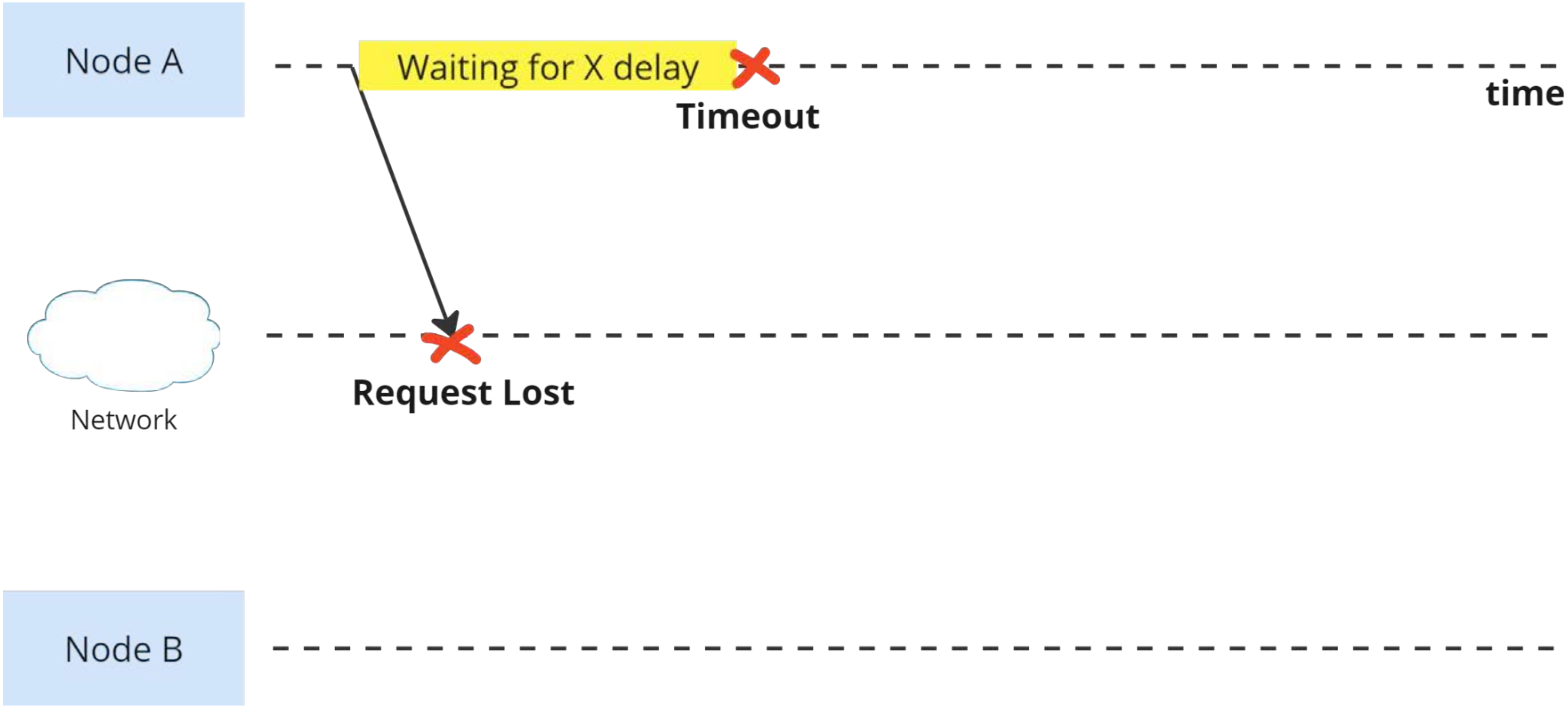
What are the main troubles?

- Unreliable Networks
- Unreliable Clocks
- Process Pauses
- Eventual Consistency
- Observability
- Evolvability

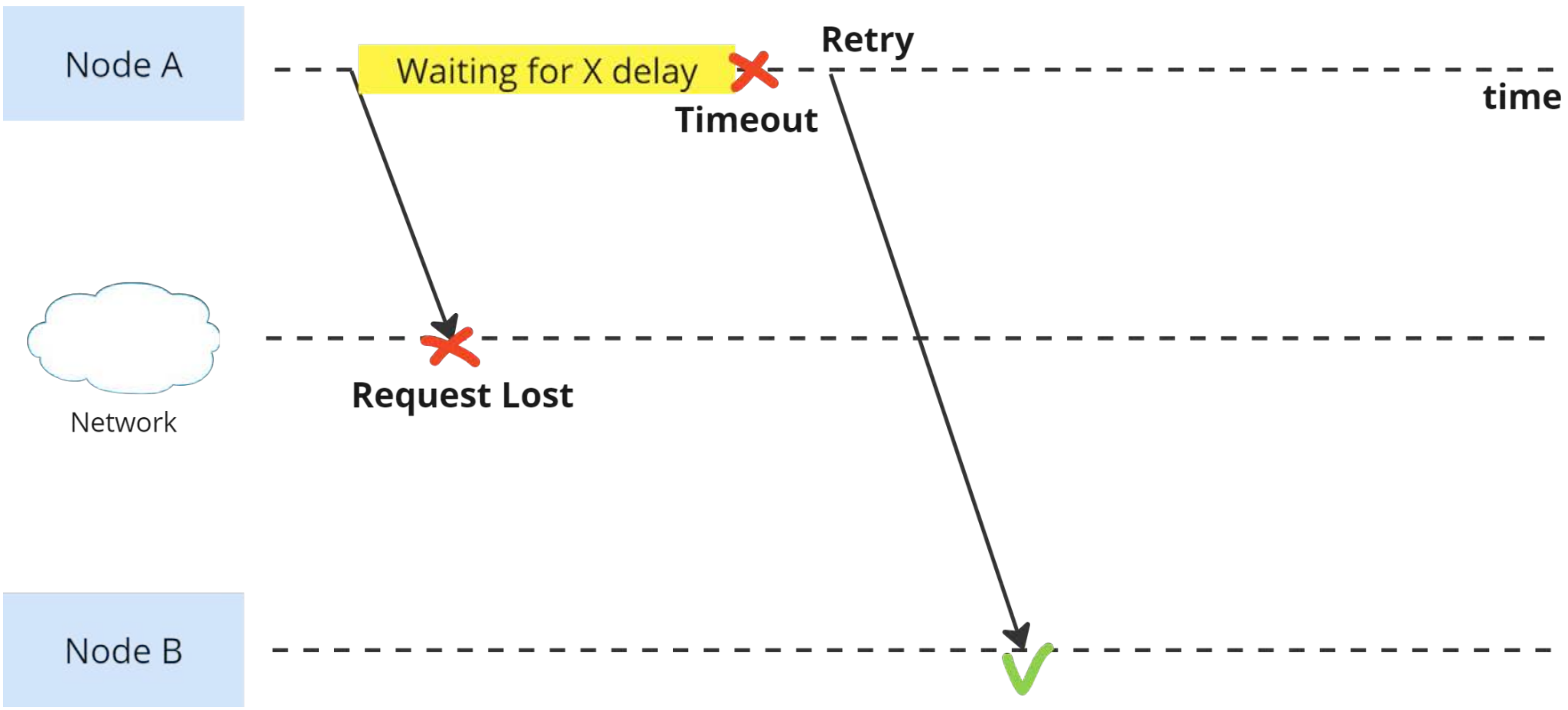
Unreliable Networks



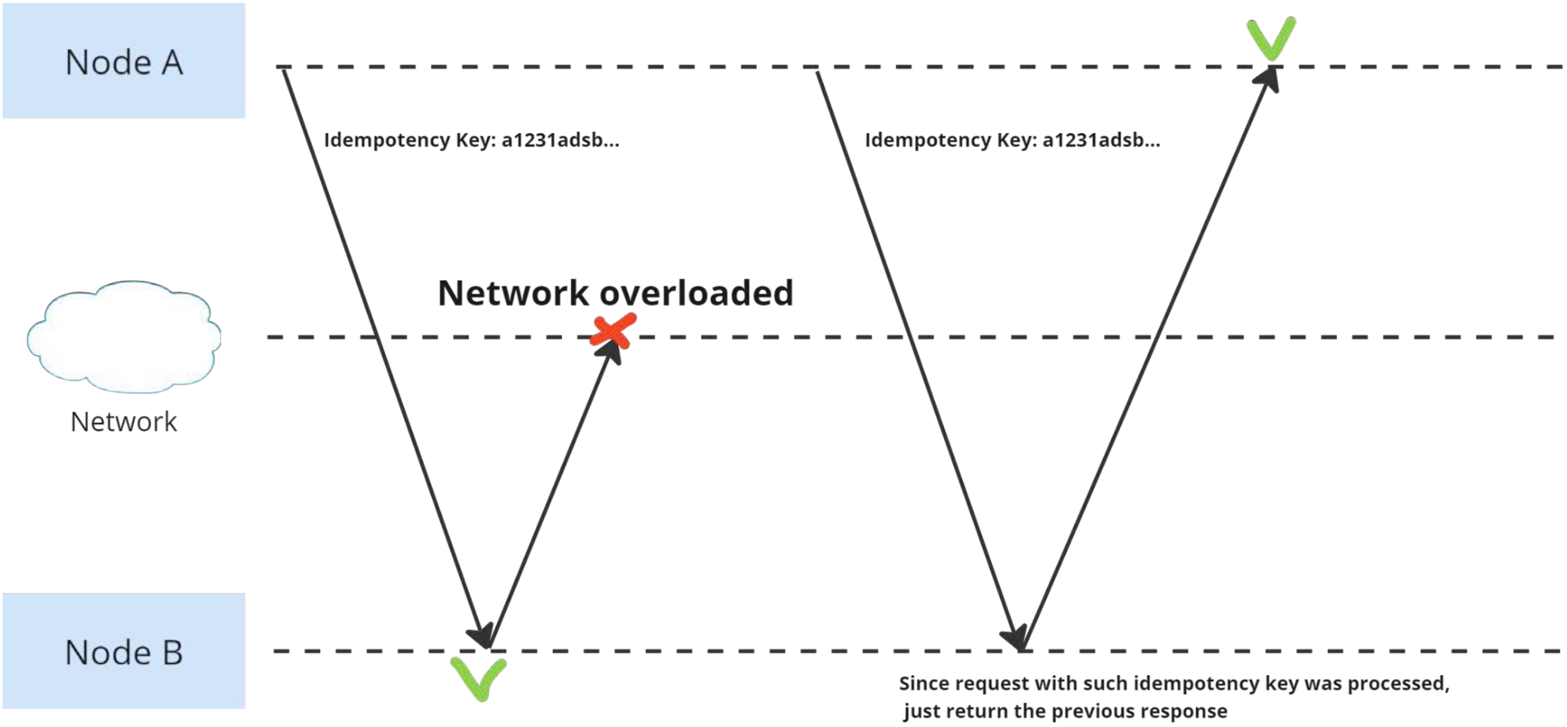
Strategy: Timeout



Strategy: Retry

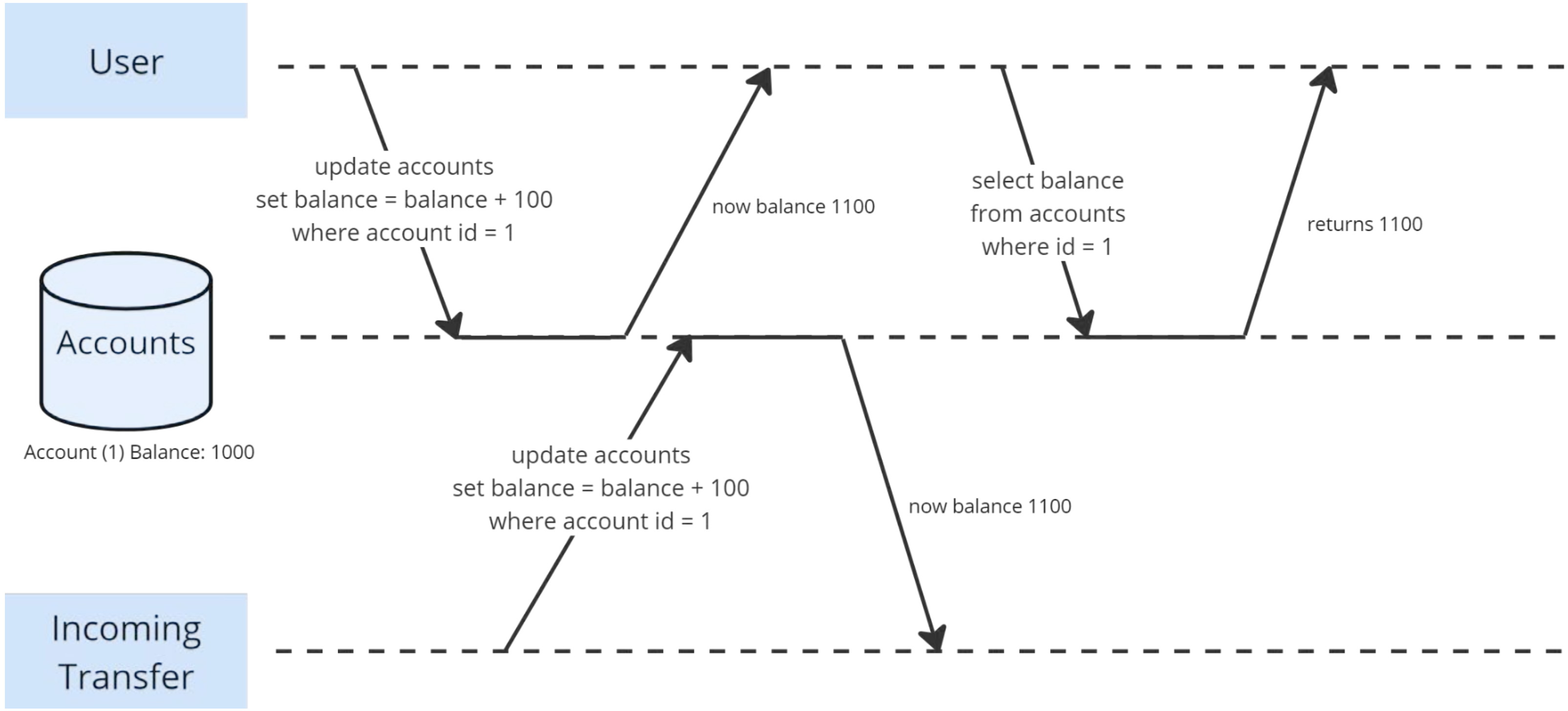


Strategy: Idempotency



Strategy: Circuit Breaker

Concurrency and Lost Writes

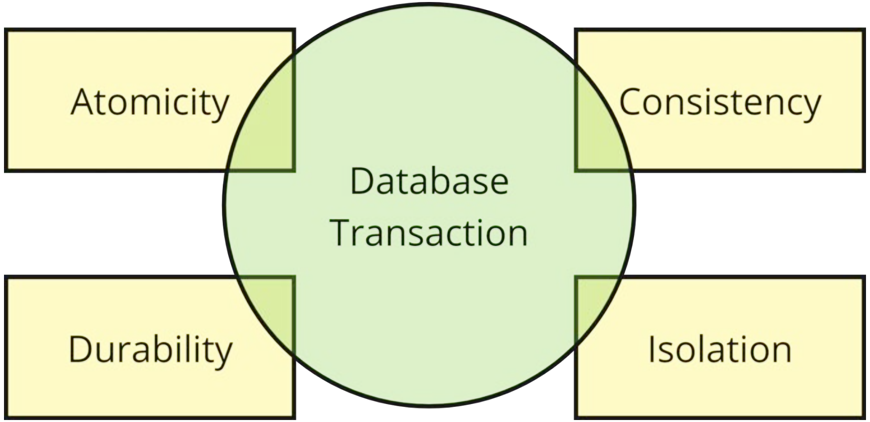


Strategy: Snapshot Isolation

All **read** operations in a transaction see a consistent version of the data as it was at the start of the transaction.

For **write** operations, changes made by a transaction are not visible to other transactions until the initial transaction commits.

Underhood it is is implemented by utilising Multiversion concurrency control (**MVCC**) approach which requires database to keep track of committed row versions and automatically detect lost writes



Strategy: Compare and Set

Cassandra Lightweight Transactions

BASE: Basically available, Soft-state, Eventually consistent

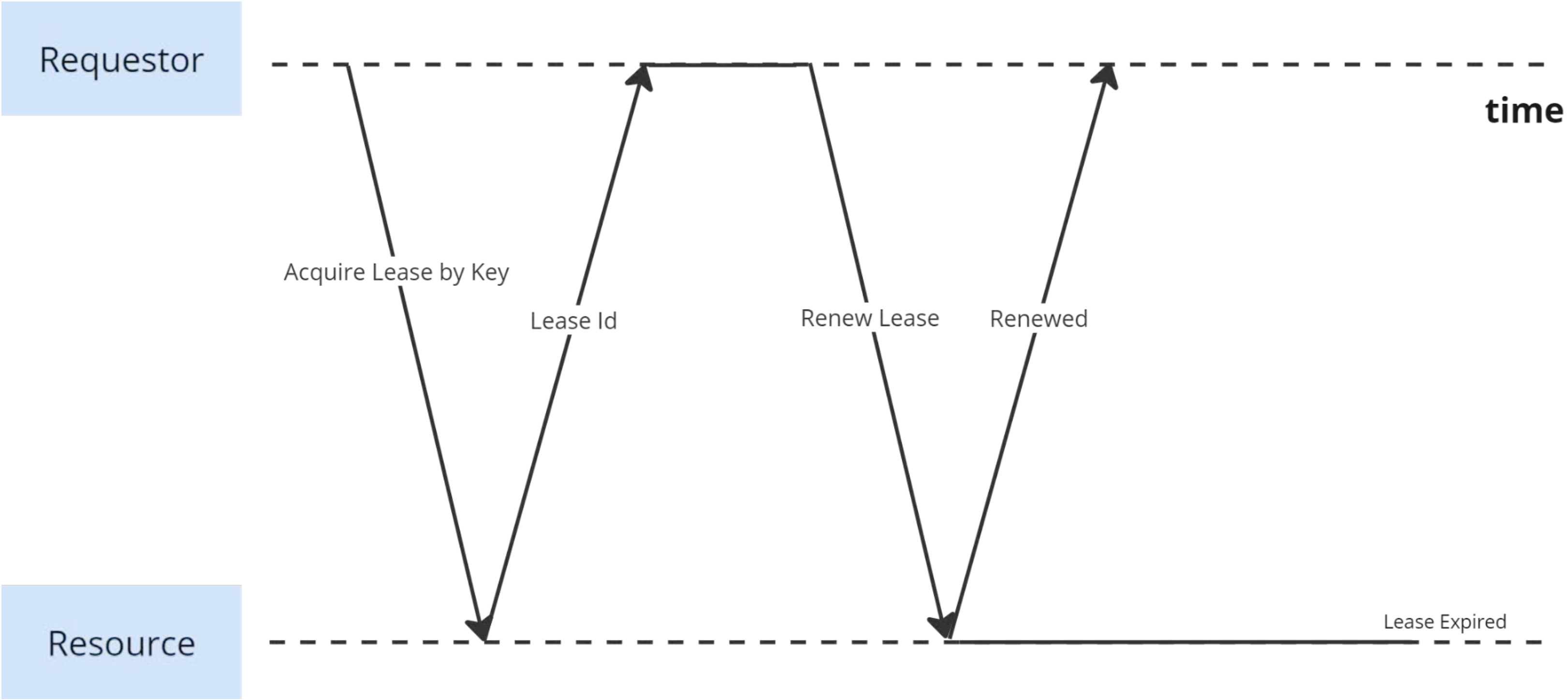
INSERT INTO [*keyspace_name*.] *table_name* (*column_list*)
VALUES (*column_values*)
[**IF NOT EXISTS**]
[**USING TTL** *seconds* | **TIMESTAMP** *epoch_in_microseconds*]

IF NOT EXISTS
Inserts a new row of data if no rows match the PRIMARY KEY values.

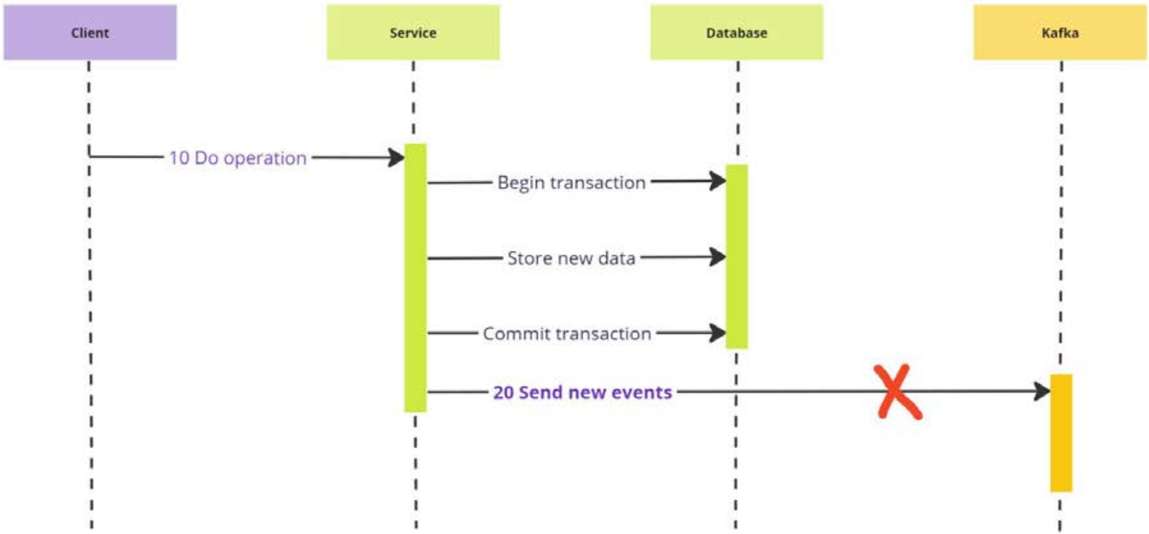
UPDATE [*keyspace_name*.] *table_name*
[USING TTL *time_value* | USING TIMESTAMP *timestamp_value*]
SET *assignment* [, *assignment*] . . .
WHERE *row_specification*
[IF EXISTS | IF NOT EXISTS | IF *condition* [AND *condition*] . . .] ;

IF
Specify one or more conditions that must test true for the values in the specified row or rows.

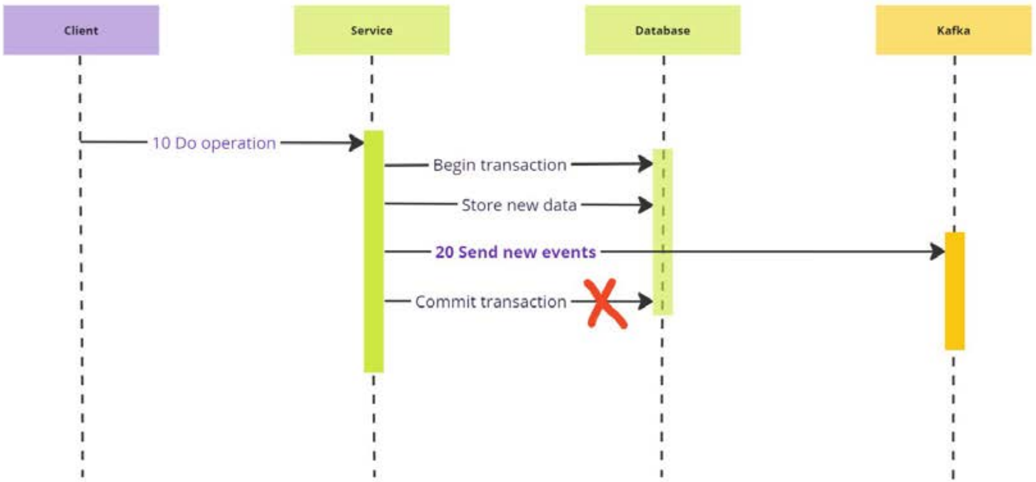
Strategy: Lease



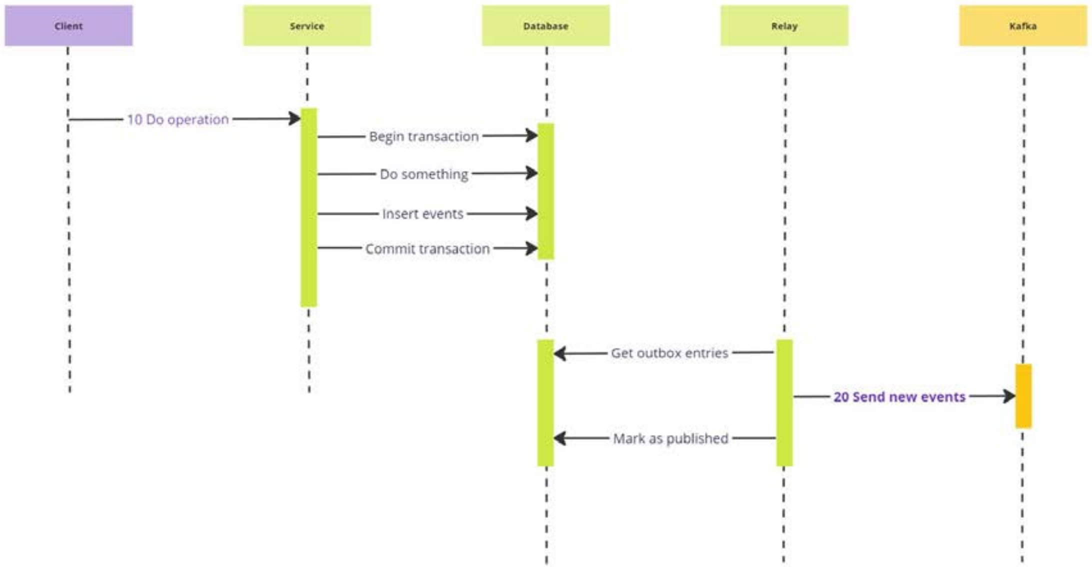
Dual Write Problem



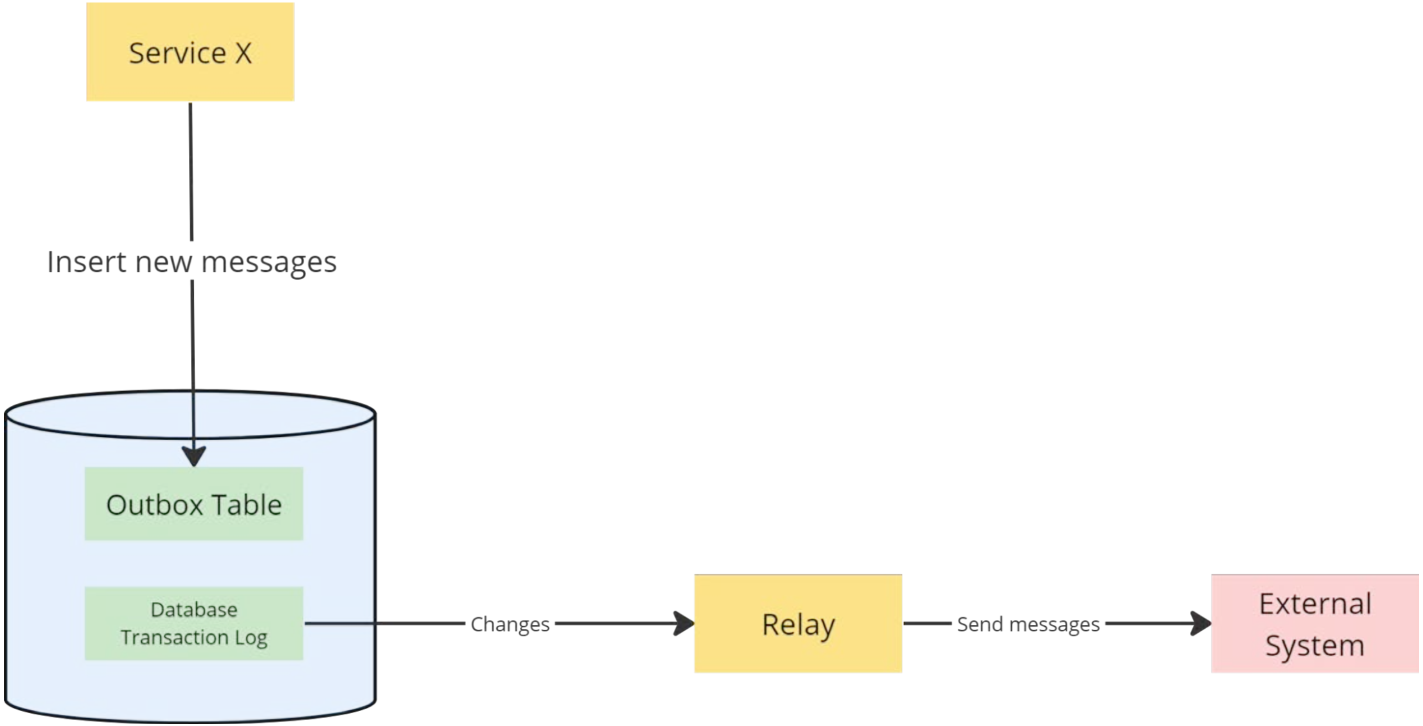
Dual Write Problem



Strategy: Transactional Outbox



Strategy: Log Tailing



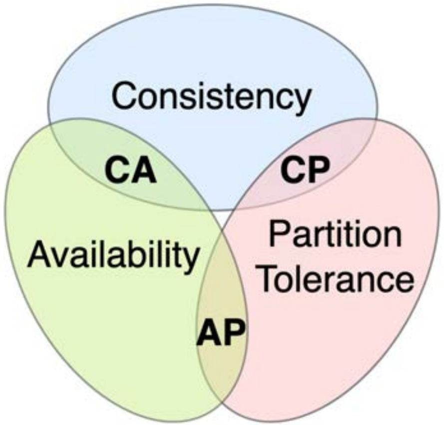
Unreliable Clocks

- Each machine has its own clock which could be faster or slower than others, depends on hardware

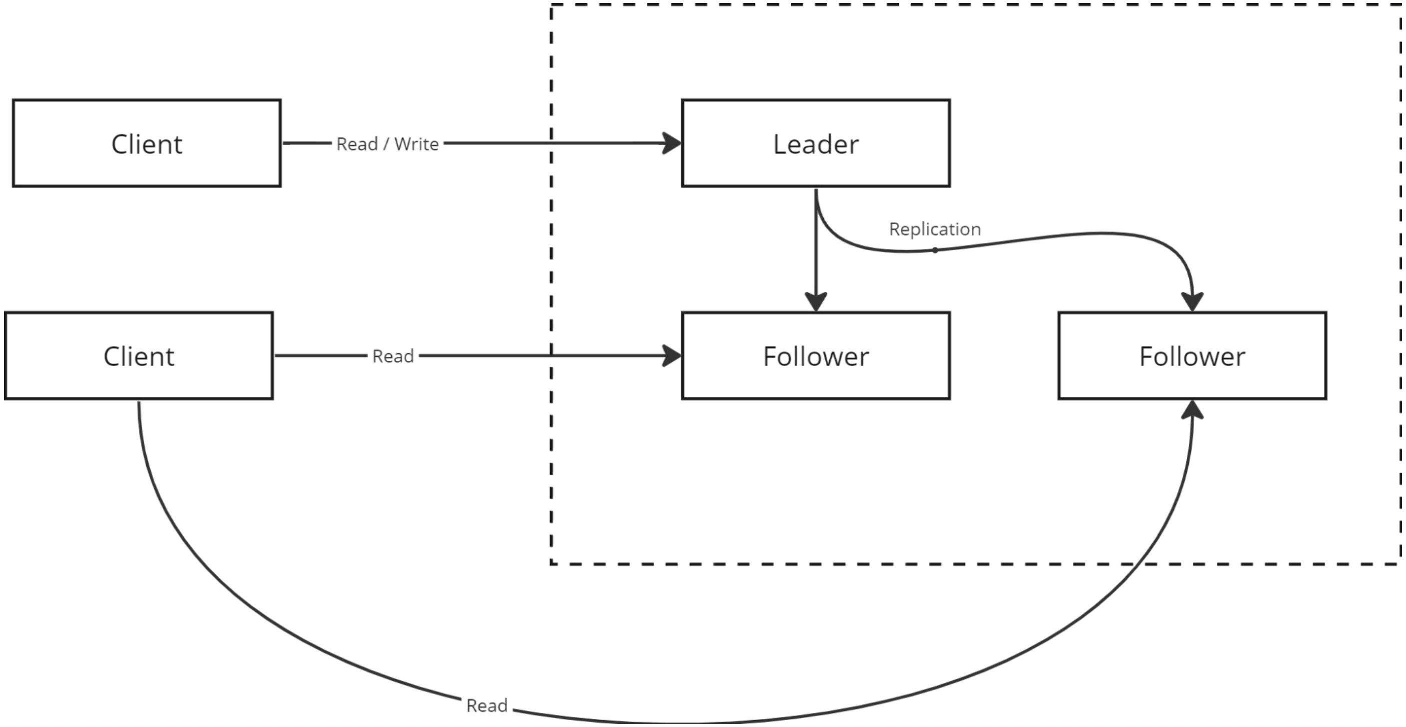
- Time-of-day clocks synchronized with NTP may be affected by network delays and issues

- Monotonic clocks are different between several computers

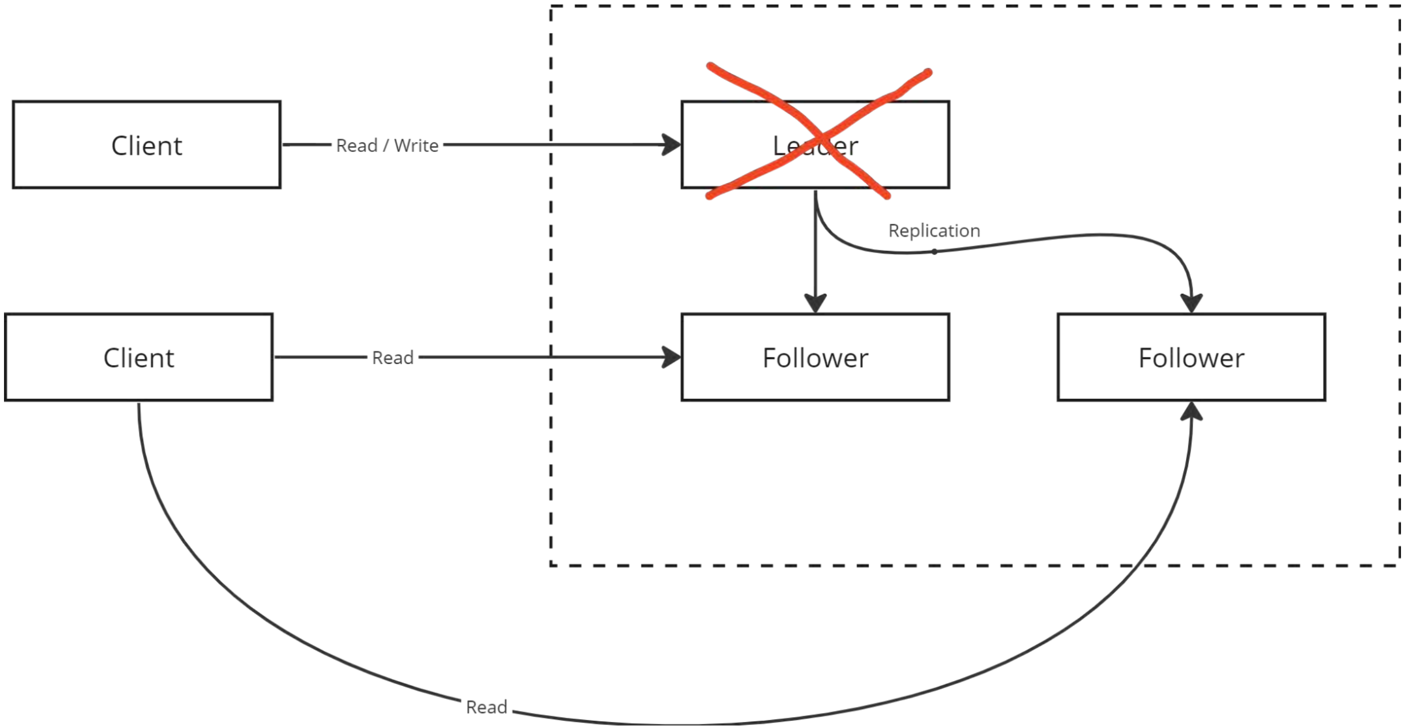
Availability and Consistency



High Availability



Failure


Client
Read / Write
Leader
Replication
Client
Read
Follower
Follower
Read

Consistency types

- Weak Consistency (Eventual consistency)
- Strong Consistency (Linearizability)

Linearizability

"In a linearizable system, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written."

- Martin Kleppmann, Designing data intensive applications

Strategy: Distributed Consensus algorithm, e.g. Raft

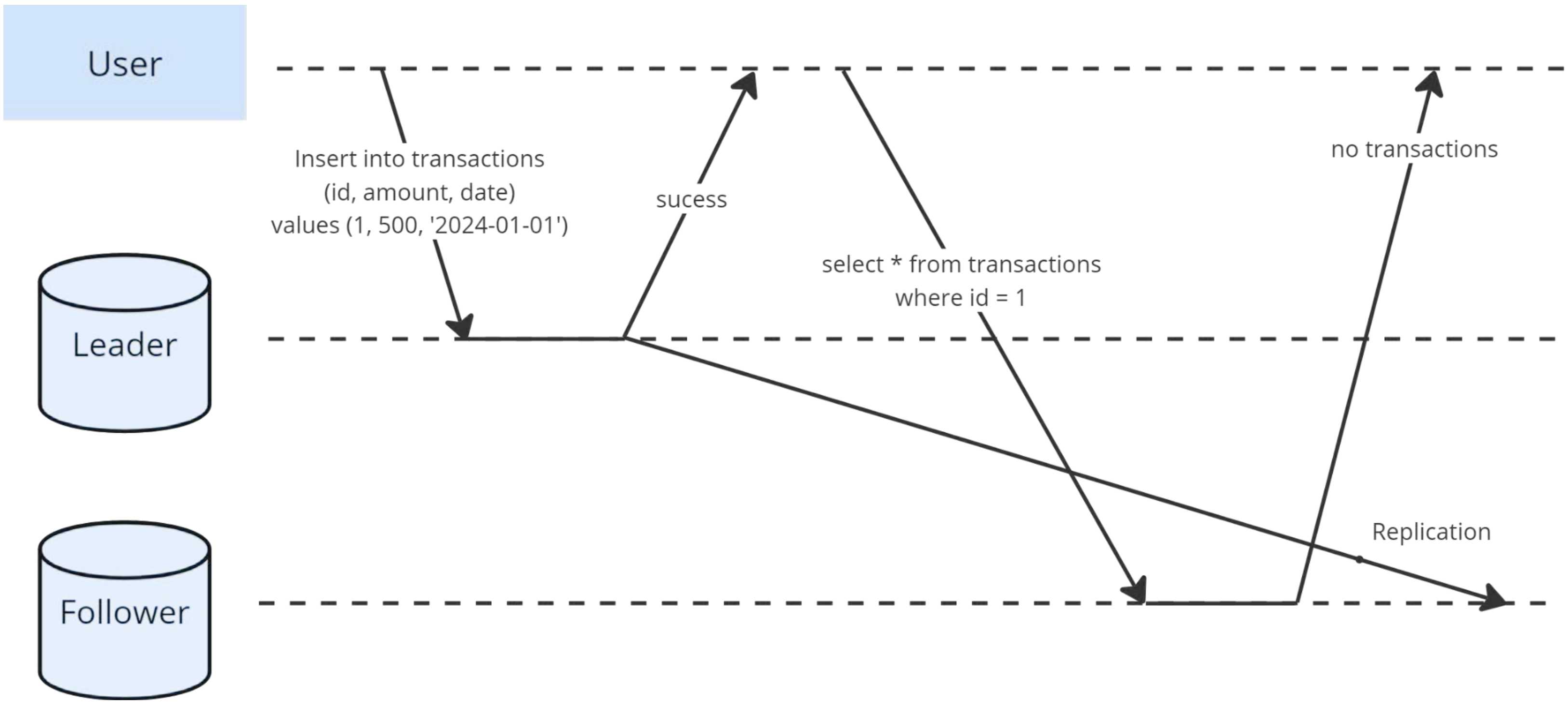
Distributed consensus is algorithm for getting nodes agree on something.

Raft is a distributed consensus algorithm. It defines Leader Election and Log Replication processes, and techniques for avoiding data inconsistency in case of networks partitions.

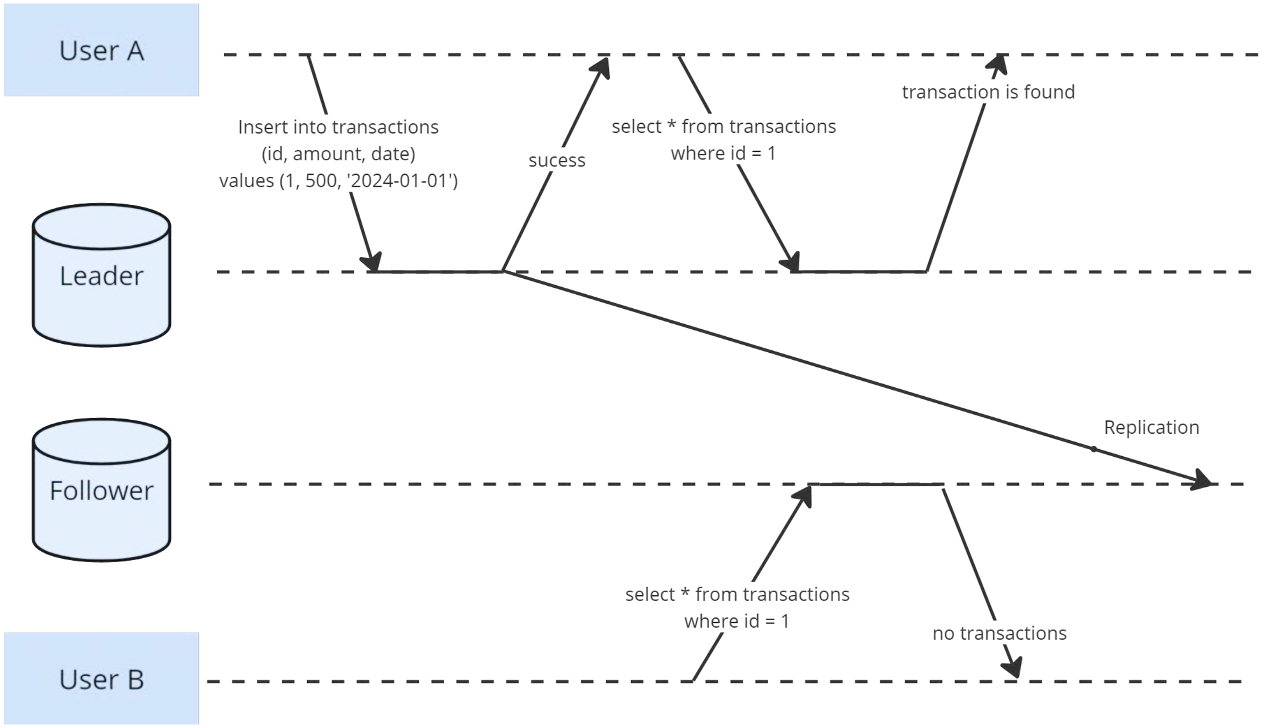
More Complexities

- Data Centers Replication
- Multi-leader replication and Conflicts resolution strategy
- Ordering, for e.g. for generating monotonically increasing number

Eventual Consistency



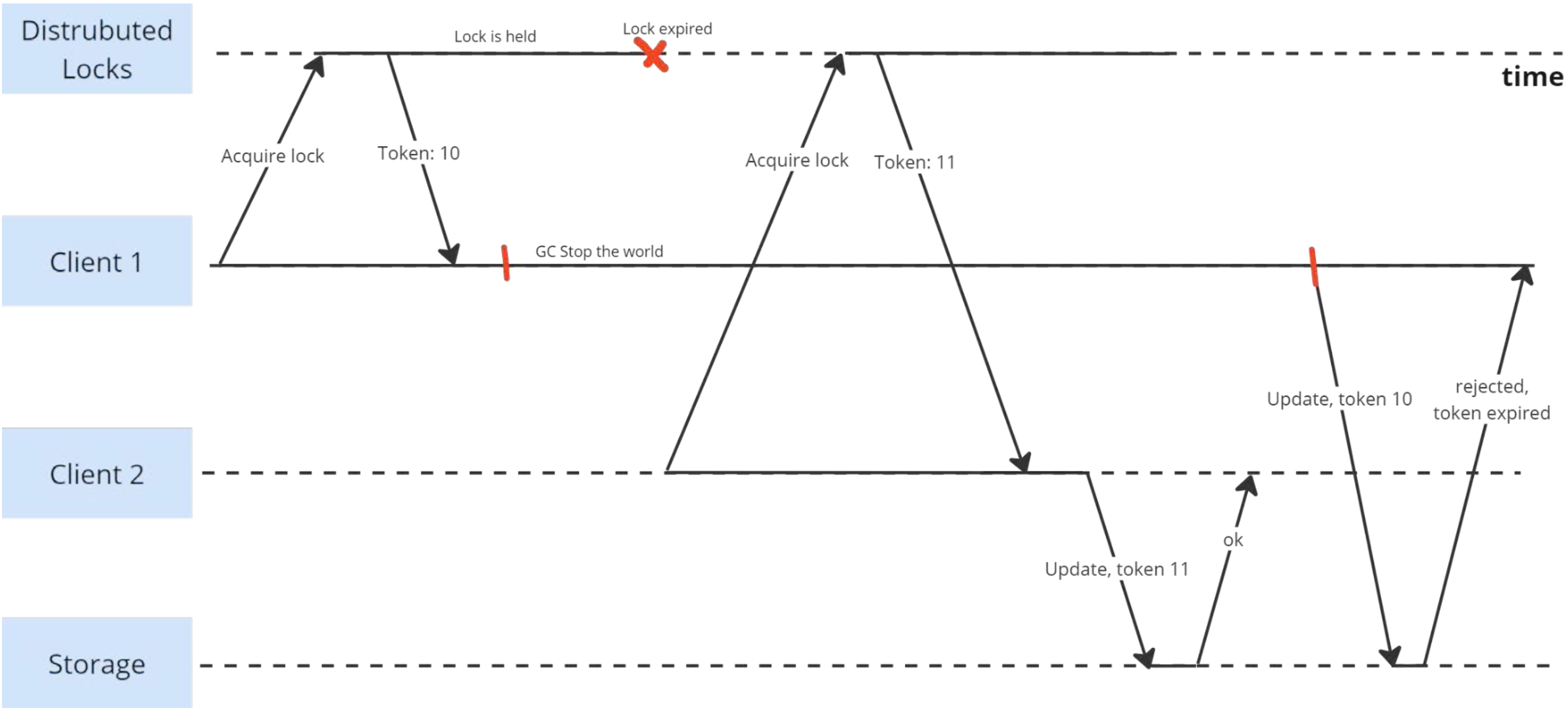
Strategy: Read from Leader



Process Pauses

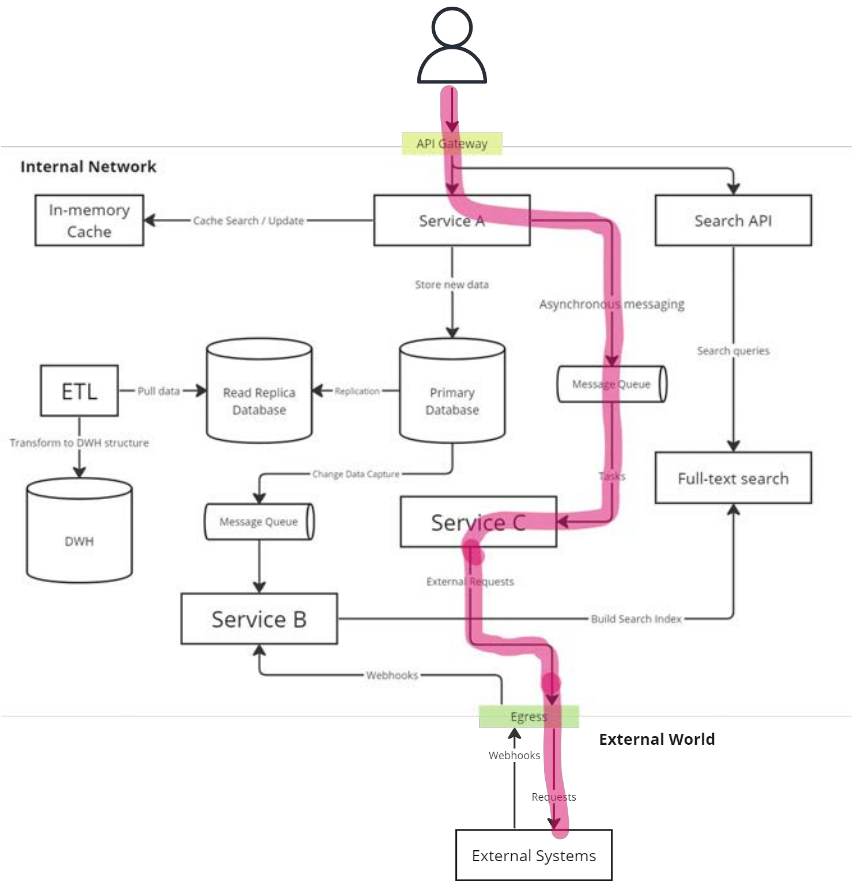
- GC (Garbage Collection) “stop the world”
- Virtual machine suspension and resume
- Context switches
- I/O delays

Strategy: Fencing



Observability

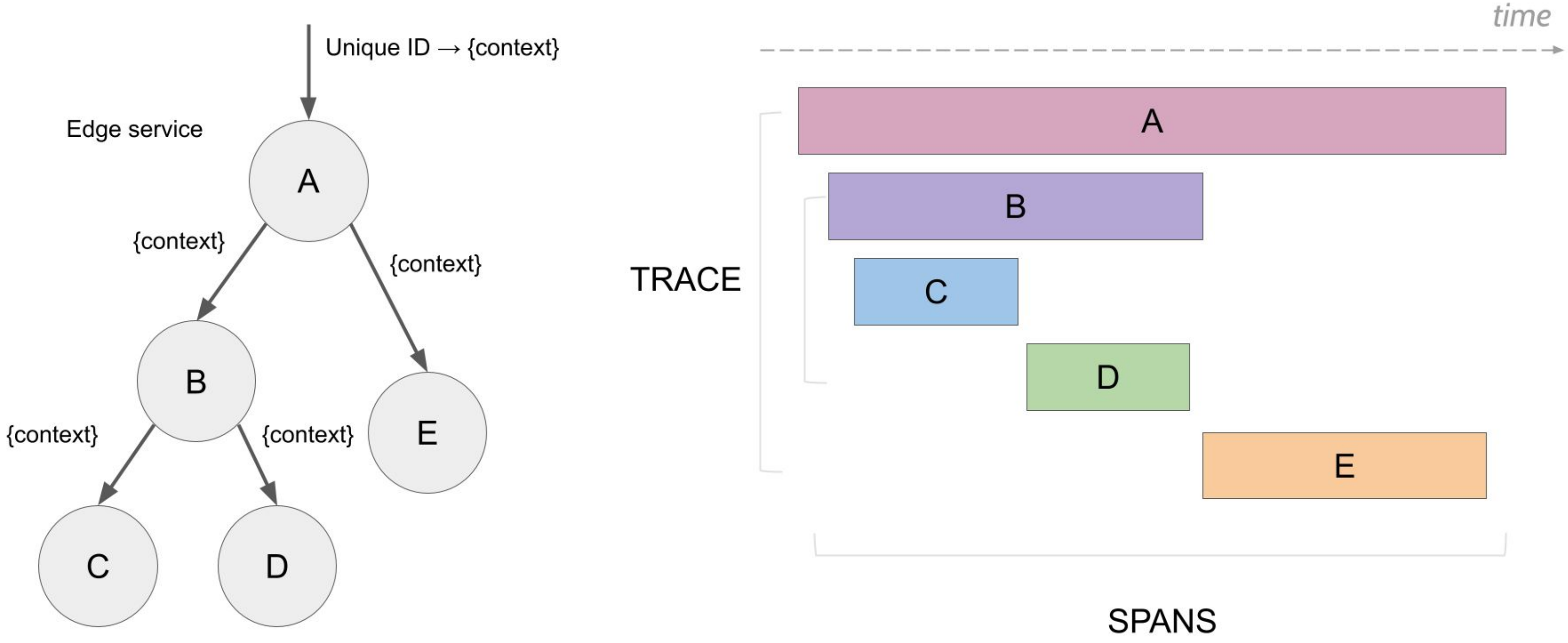
Example journey across multiple components



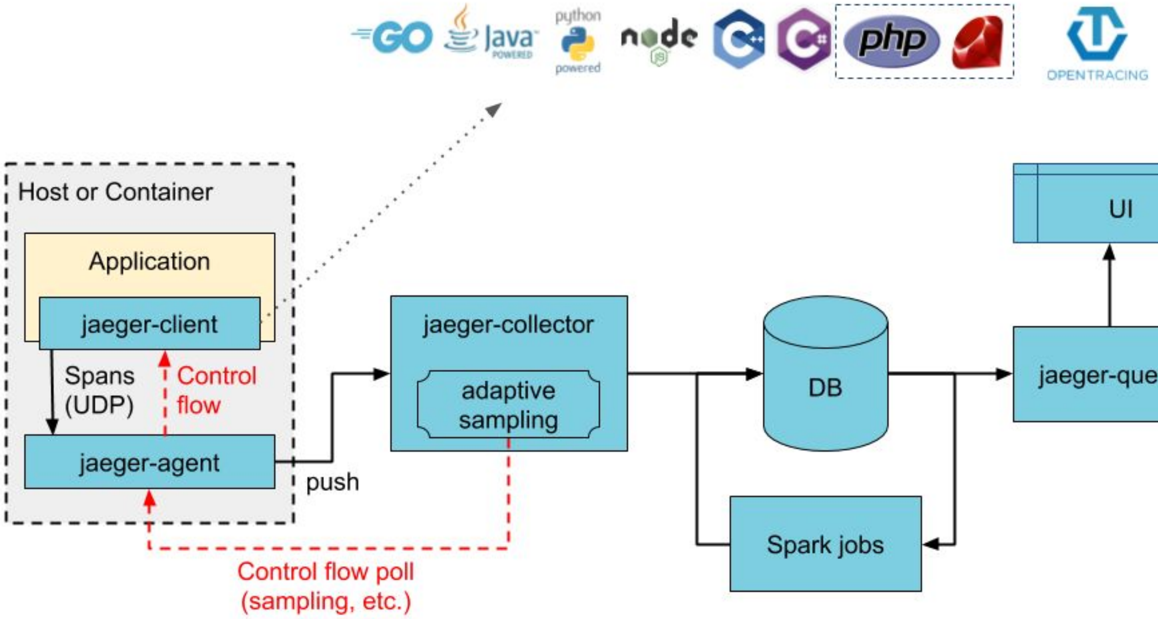
Challenges:

- Understanding system interactions
- Performance optimisation
- Root cause analysis
- Cost efficiency

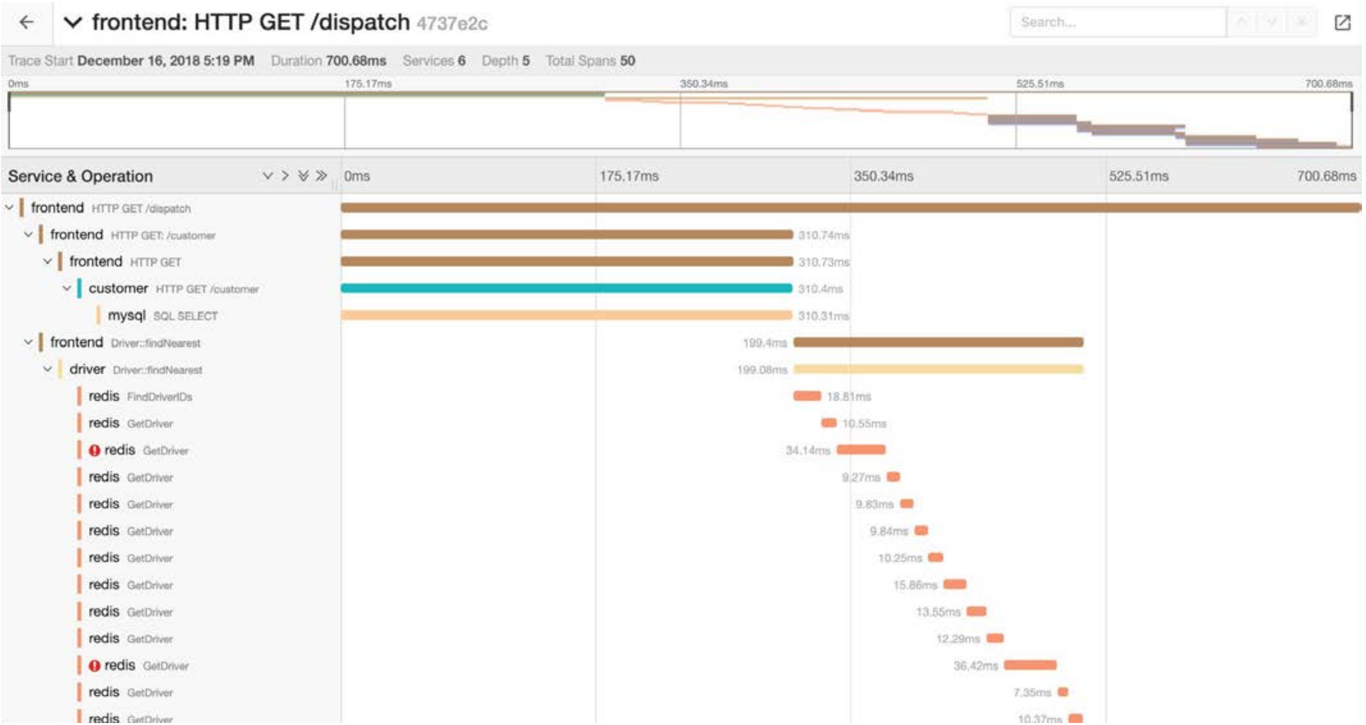
Strategy: Distributed Tracing



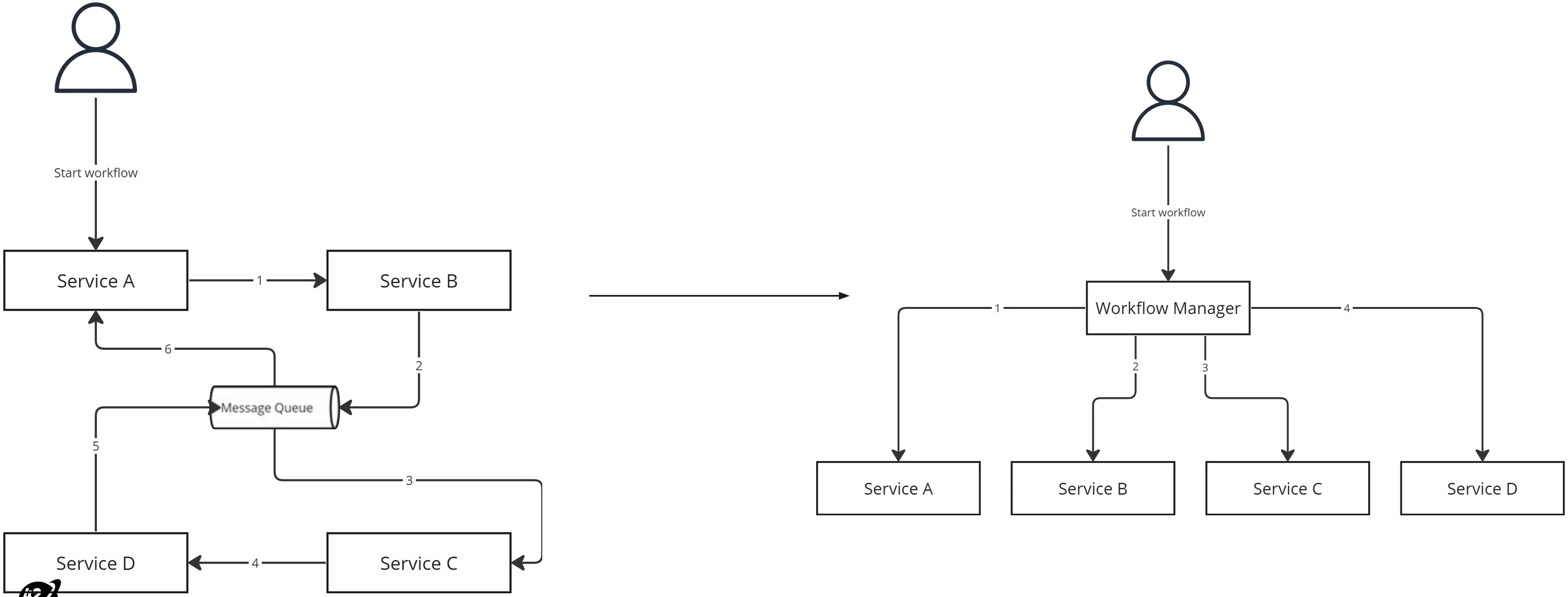
Strategy: Distributed Tracing



Strategy: Distributed Tracing



Strategy: Orchestration over Choreography



Evolvability and Cybernetics principles

- System Thinking
- Feedback Loops
- Adaptability and Learning
- Goal-oriented design
- System Hierarchy

Systems Thinking

This concept focuses on the system as a whole rather than its individual parts. In software engineering, this means considering how all parts of a software system (e.g., modules, functions, infrastructure) work together to achieve the desired outcomes.

Design decisions are made with an understanding of their impact on the entire system.

Example: When Service B handles an event published by Service A, the outcome does not affect Service A directly. However, the overall result of the operation is significant to the system as a whole.

Feedback Loops

A core concept in cybernetics is the use of feedback loops to control and stabilize systems.

Example: In software architecture, feedback loops can be implemented in various forms, such as monitoring system performance, user feedback mechanisms, or continuous integration/continuous deployment (CI/CD) pipelines.

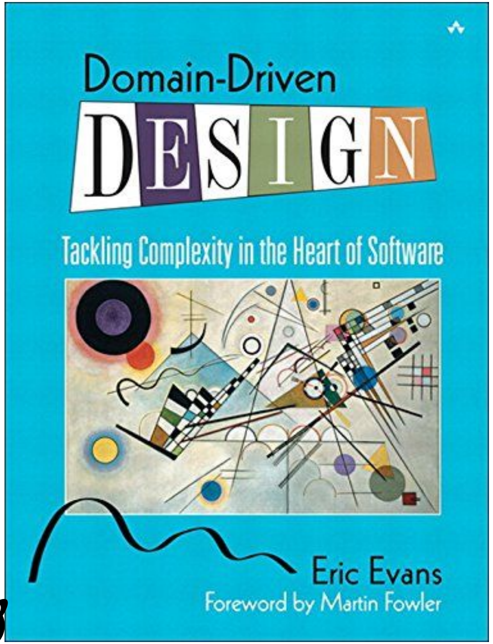
Adaptability and Learning

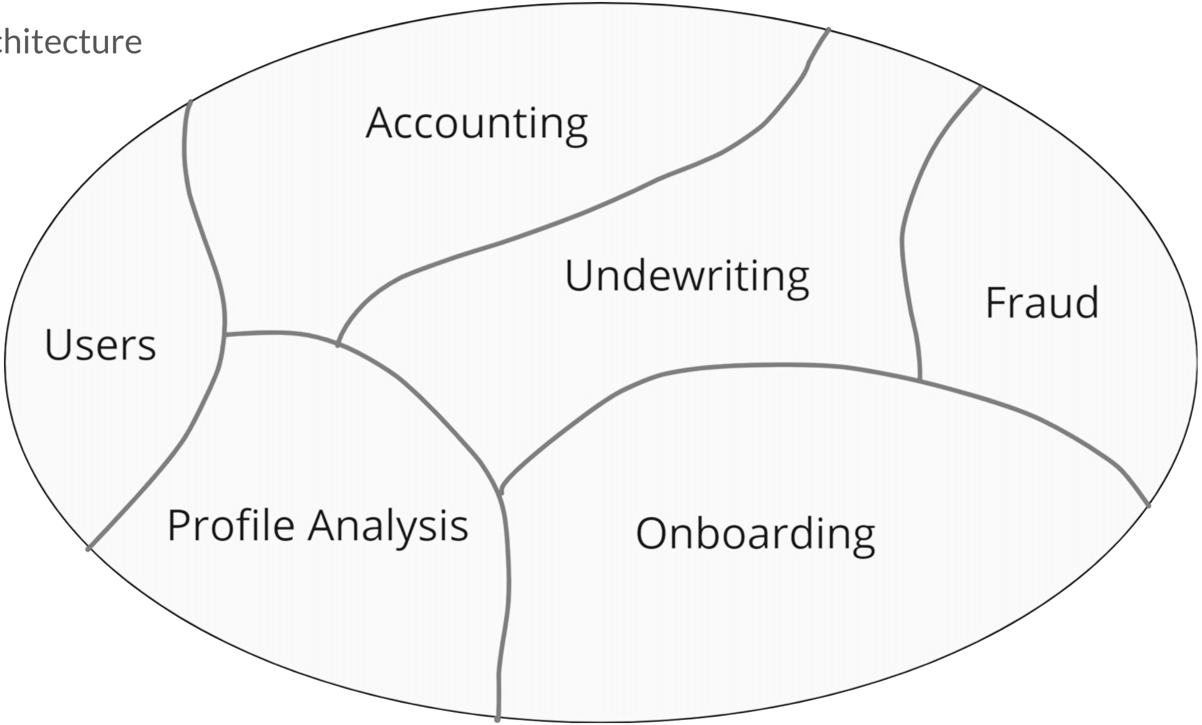
Cybernetics promotes the idea that systems should be capable of adapting to changes in their environment. For software architecture, this means designing flexible systems that can evolve over time.

Example: This could involve using microservices that can be updated independently, employing feature toggles for managing new features, or incorporating machine learning algorithms that improve with more data.

Goal-oriented design

Business Requirements drives Architecture





Big Ball Of Mud

Hierarchy

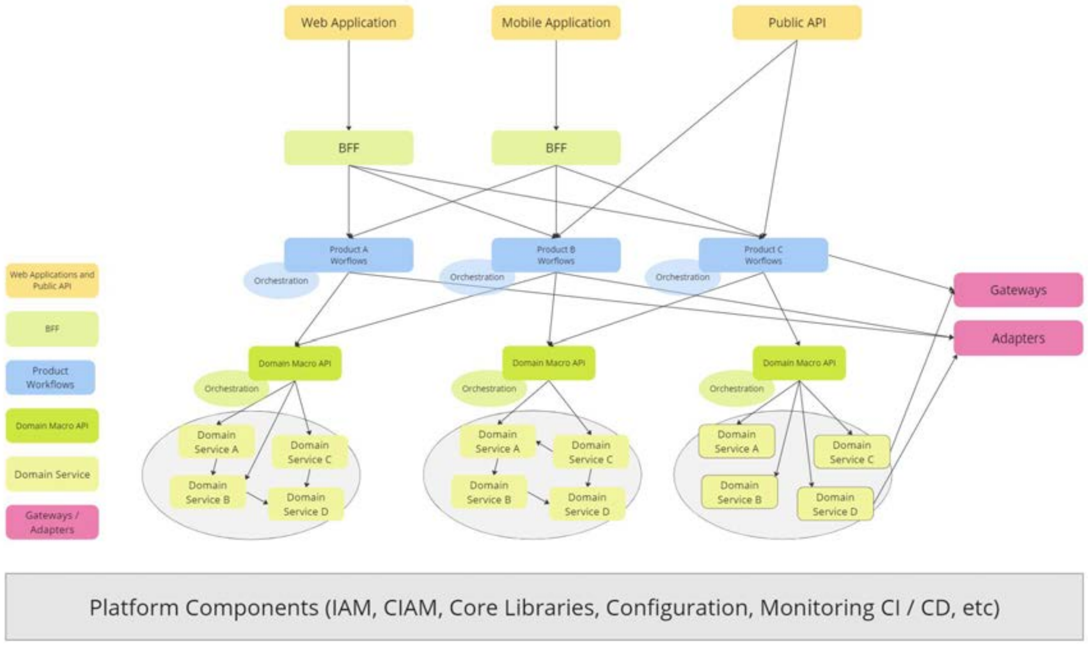
- Systems are organized in hierarchies of subsystems.

- Software systems often have a hierarchical structure, with high-level modules depending on lower-level modules for functionality.

- This hierarchical decomposition helps manage complexity by breaking down the system into more manageable parts.

Fallacy: All microservices are the same

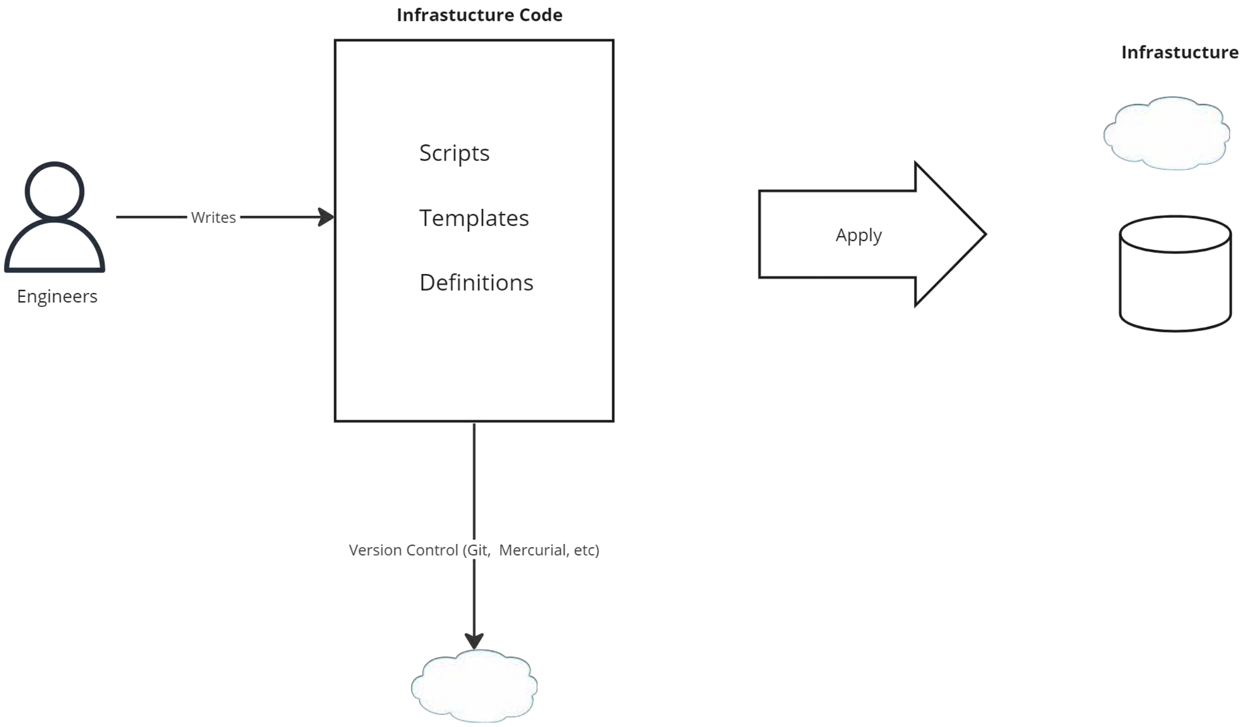
Strategy: Service Types



SRE Principles

- Embrace the risk
- Use SLA, SLO, SLI to define system reliability
- Automate manual work
- Monitor everything
- Simplify as much as you could

Infrastructure as Code



Chaos Engineering and Testing: Jepsen tests

Jepsen is a tool and a framework developed by Kyle Kingsbury to analyze the safety and consistency of distributed databases and systems under various conditions, particularly focusing on how these systems behave under network partitions and other types of failures.

Fault Injection: Jepsen introduces faults into distributed systems to observe how they behave under failure conditions. This includes network partitions, where communication between nodes in the system is deliberately severed.

Operations Testing: It tests various operations such as reads, writes, updates, and deletions across different nodes to see if the system maintains consistency.

Concurrency: Jepsen tests systems under concurrent operations to simulate real-world usage where multiple processes may interact with the system simultaneously.

Simplicity and Measuring Complexity

"A complex system that works is invariably found to have evolved from a simple system that worked."

- Gall's Law

- Cyclomatic complexity
- Time to train
- Explanation time

Thank you for attending!