FLEX - Data Integration

Simon Härer, Michael Sperber

Created: 2024-06-10 Mon 07:48

How to communicate?

Now that we have stand-alone services, nicely modularized, they need to communicate in some way to share their domain-specific data.

Exercise: Integration and Communication

How do microservices or SCS communicate and share data?

- algorithmically
- technologically

Brainstorm! Discuss trade-offs!

Integration and Communication

There are several methods and technologies that can be used to integrate and communicate:

- Data Replication
- REST
- RPC
- Messaging
- (Frontend)

Each method comes with different aspects of coupling.

Asynchronous Replication via ETL Tools

"Extract, transform, load"

- "Integrators" shipped with many database products, hook into transaction log
- Apache Spark - offline batch processing or streaming-driven
- NiFi - offline, graphical data routing

Synchronous Replication within Databases

Replication is synchronization of a database between multiple nodes. This is convenient, as it creates the illusion of a single source of truth.

Synchronous Replication within Databases

Synchronous Replication as an integration mechanism has disadvantages:

- The data schema is shared among multiple clients. Clients tend to use the specific data schema they require (cf. Bounded Contexts)
- Changes in schema affect all clients that use it
- Thus, strong coupling!

Replication: Database Transaction Isolation Levels

Isolation is typically defined at database level as a property that defines how/when the changes made by one operation become visible to other.

Take the following example: An order in an online shop consists of meta information and line items. If the meta-information was written, but the line items not yet, is the meta-information visible to other users?

ACID

ACID: atomicity, consistency, isolation, durability

- Set of properties for database transactions, for guaranties in case of errors
- Classical database transaction fullfils ACID
- **Atomicity**: A transaction composed of multiple operations is treated as a single unit

CAP Theorem

The CAP theorem states that, for a distributed data store, we can only have two of the following properties:

- **Consistency**: Every read results in the most recent write
- **Availability**: Every request receives a response
- **Partition tolerance**: The system operates despite an arbitrary number of messages being dropped by the network

Starke, Gernot. *Effektive Softwarearchitekturen: Ein praktischer Leitfaden*. Carl Hanser Verlag GmbH Co KG, 2015.

Network Partition Failure

When a network partition failure happens, the following choice must be made:

- Either cancel an operation, this results in consistency but violates availability
- Or proceed and keep up availability and maybe sacrifice consisteny

Replication: Database Transaction Isolation Levels

The following levels exist:

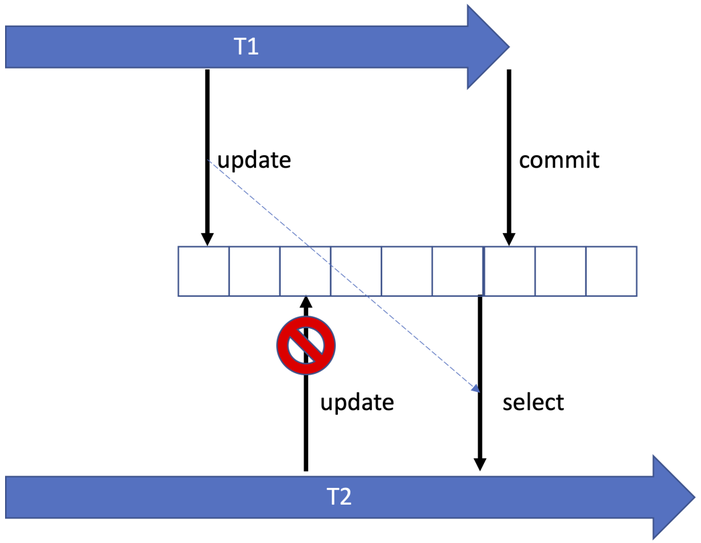
- Serializable
- Repeatable reads
- Read committed
- Read uncommitted

(Serializable is almost never the default.)

Wikipedia

Read uncommitted, dirty reads

Dirty reads are allowed: one transaction may see not-yet-committed changes made by other transactions.



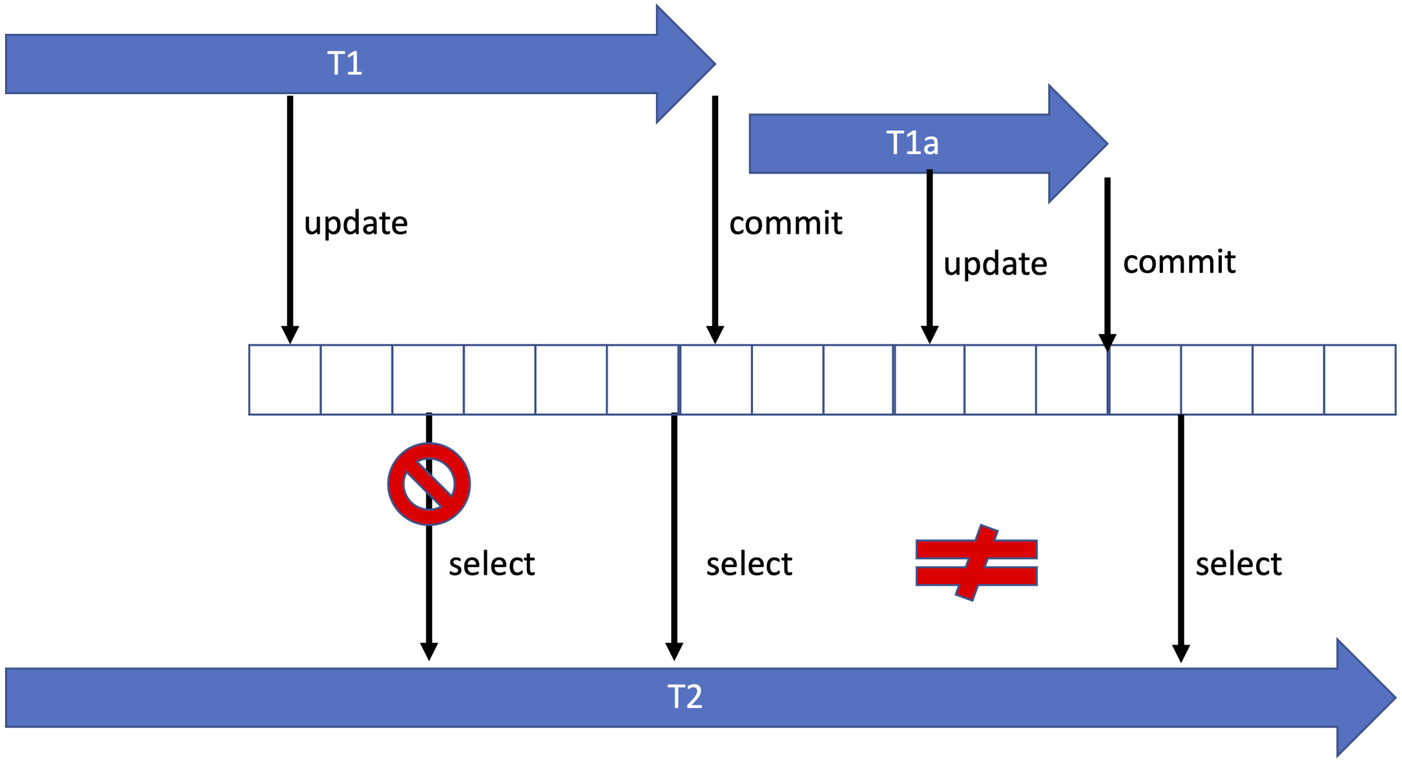
Read committed

"write locks"

Any data read is committed at the moment it is read.

If the transaction re-issues the read, it may not find the same data; data is free to change after it is read.

Read committed, unrepeatable reads

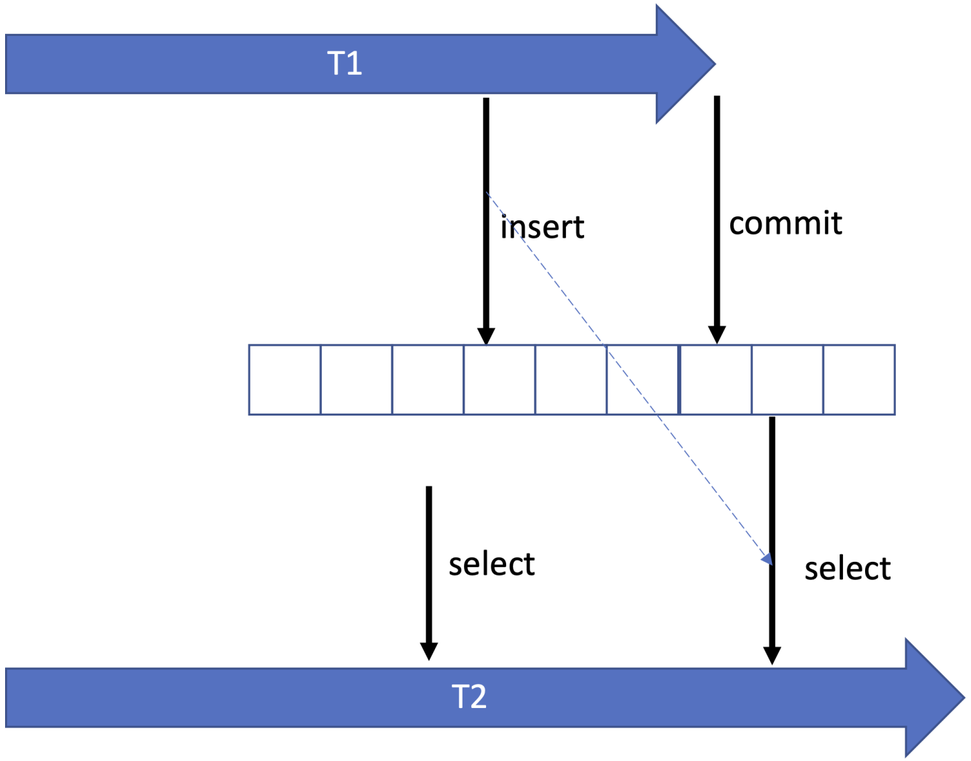


Repeatable reads

"read and write locks"

Two writes are allowed to the same column(s) in a table by two different writers (who have previously read the columns they are updating), resulting in the column having data that is a mix of the two transactions.

Repeatable reads, phantom reads



Serializable

A serializable execution produces the same effect as some sequential execution of those same transactions.

Example: Oracle DB

Oracle's default transaction isolation level is read committed. That is, when two consecutive reads on the same data occur within the same transaction, the data could differ if another transaction was committed in between. The phenomenon is called non-repeatable reads. In the case of newly inserted data, that "appears" out of nowhere in the second read, we talk about phantoms.

If this default is not known to the developer it can cause serious bugs.

The serializable isolation level cannot be applied in Oracle in a distributed setting!

Jepsen

"Jepsen is an effort to improve the safety of distributed databases, queues, consensus systems, etc. We maintain an open source software library for systems testing, as well as blog posts and conference talks exploring particular systems' failure modes. In each analysis we explore whether the system lives up to its documentation's claims, file new bugs, and suggest recommendations for operators."

Jepsen vs. MongoDB

"A brief investigation into MongoDB 4.2.6's transaction system found violations of snapshot isolation, rather than claimed "full ACID" guarantees. Weak defaults allowed transactions to lose writes and allow stale reads unless carefully controlled."

MongoDB 4.2.6, Kyle Kingsburgy, 2020-05-15

Distributed transactions: two-phase commit

With two-phase commits (2PC) we can guarantee atomicity

- Participants (database replicas) can give a transaction commit to vote
- Participants can vote commit or abort (e.g. if the transaction would violate ACID guarantees)
- Coordinater coordinates votes and if all vote for commit the transaction is applied on at all participants, else not.

Wikipedia, *Two-phase commit protocol*

ACID & distributed transactions are often not required. They increase complexity and performance suffers.

Replicated Databases

- Cassandra: Geographically distributed , own query language
- HBase: NoSQL, on top of Hadoop Distributed Filesystem
- SimpleDB: Cloud-based, low operational overhead
- Couchbase Server: NoSQL document database
- ...

REST

Representational State Transfer

- Synchronous
- Stateless communication with HTTP (no state of the client stored on the server)
- Each resource can be created/retrieved/updated/deleted by a globally unique URI

REST: HATEOAS

Hypermedia as the Engine of Application State

- Relationship between resources is modelled by links
- In theory, client just needs to know an entry point
- Implementation via HAL: Links organized in JSON
- Synchronous

Wolff, Eberhard. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag, 2018.

SOAP & Remote Procedure Call (RPC)

- Calls a specific procedure on a server (RPC) based on an interface specification
- Encryption and redirection using WS-Security standard
- Many different extensions available
- Synchronous

SOAP is more sophisticated than REST but needs strong coordination e.g. when business logic is changed. Even small changes in the interface specification must be propagated to all teams.

Wolff, Eberhard. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt. verlag, 2018.

Messaging

Microservices communicate by sending messages to each other.

- Messages can be broadcast, so that the sender doesn't even have to know the receiver exists
- Messages often describe commands or events
- Transactions can be modelled using Two Phase Commits (2PC)
- Asynchronous

REST/RPC vs. Messaging

Both REST/RPC and Messaging require handling errors due to distribution.

Synchronous communication most closely approximates local procedure calls, and situations where a client needs the response to a request before moving on

However:

- Synchronous communication introduces temporal coupling
- Synchronous communication (REST/RPC) hampers parallelism

Messaging decouples "requests" from "responses":

- Messaging encourages loose temporal coupling
- More effort to implement "request-response" pattern

Messaging Patterns

- **Request–reply** connects a set of clients to a set of services. This is a remote procedure call and task distribution pattern.
- **Publish–subscribe** connects a set of publishers to a set of subscribers. This is a data distribution pattern.
- **Push–pull** connects nodes in a fan-out / fan-in pattern that can have multiple steps, and loops. This is a parallel task distribution and collection pattern.
- **Exclusive pair** connects two sockets in an exclusive pair. This is a low-level pattern for specific, advanced use cases.

Wikipedia, *Messaging Pattern*

Message-Oriented Middleware

For messaging various technologies exist, that provide certain guaranties and features. They are used as a middleware.

- **AMQP** (Advanced Message Queuing Model) is a protocol for message exchange. It defines a method for coordination (link protocol), types, formats. A popular implementation is RabbitMQ.
- **Kafka** focuses on persistence, failure safety and replication. Categorizes messages by topics.

Message-Oriented Middleware

- **ZeroMQ** comes as a library and not as a server. This makes it very lightweight and easier to deploy. It is based on methodologies as using pub/sub, push-pull, or client-server patterns.

Other mentions are Java Messaging Service or ATOM.

Events

- object that describes something that happened **in the past**
- should be based on domain rather than implementation

vs. **Commands**:

- object that describes a request that something happens **in the future**
- should be a separate type from events

Event Sourcing

Event sourcing stores events denoting changes:

- Only events are stored
- Data can be reconstructed by aggregating a history of events
- These projections are an interpretation of truth
- Projection of state can be different for every service

Ben Stopford, *Designing Event-Driven Systems*, O'Reilly Media, 2018

Event Sourcing

- Conflicts can easily be recognized, without losing information
- Free audit log
- Bug reports are easy to replay

Event-Driven Architecture - Events

- The system is modelled based on events, their production, detection & consumption
- State changes are always mapped to events
- Interaction between event emitters, channels and consumers
- Emitters do not know who consumes their events

Ben Stopford, *Designing Event-Driven Systems*, O'Reilly Media, 2018

Event-Driven Architecture - Commands

- Often in combination with *commands* - e.g. a user clicks on a button which produces a command. Commands are processed, generating events - possibly zero or more than one.
- Supports loose coupling of components

Commands vs. Domain Events

- command: what someone **wants** to happen
- event: happened in the past
- event: describes side effect
- domain event: refers to ubiquitous language of DDD

Transactional vs. Data-Flow-Oriented Systems

Transactional: **state**, "now"

Data flow: **facts**, "eventually"

Modularization vs. System Type

system type	preferred modularization	consistency guarantees
transactional consistency	single process	strong
data-flow	single executable	fast eventual
	message bus	medium eventual
batch	arbitrary	eventual

Modularization vs. System Type

In principle, coordinating transactional consistency across a distributed system is possible, but very difficult, error-prone, and generally does not perform well. (Cf. section on Distributed Computing.) This possibly limits the choice of modularization type.

Data-flow systems are easier to distribute, and thus more agnostic to modularization type.

Data-flow systems can also be structured as *batch systems*, with only loose temporal coupling, further widening the choice of

modularization type. However, batch systems also give looser guarantees.

BASE

BASE: Basically Available, Soft state, Eventual consistency

In contrast to ACID!

- **Basic Availability**: basic read & write operations are available as much as possible, however, without consistency guarantees
- **No atomicity, no isolation**: follows from the lack of consistency guarantees

Feeds

Feeds are a good choice if a split happened along the customer journey because data sharing is often along this uni-directional flow.

- Application offers feed with domain data, e.g. a stream of events
- Consuming Applications just consume the data they need
- Pull mechanism leads to asynchronous communication
- In theory, a feed provider does not care who consumes and why

Exercise: Data Integration

- Can we achieve transactional consistency between systems?

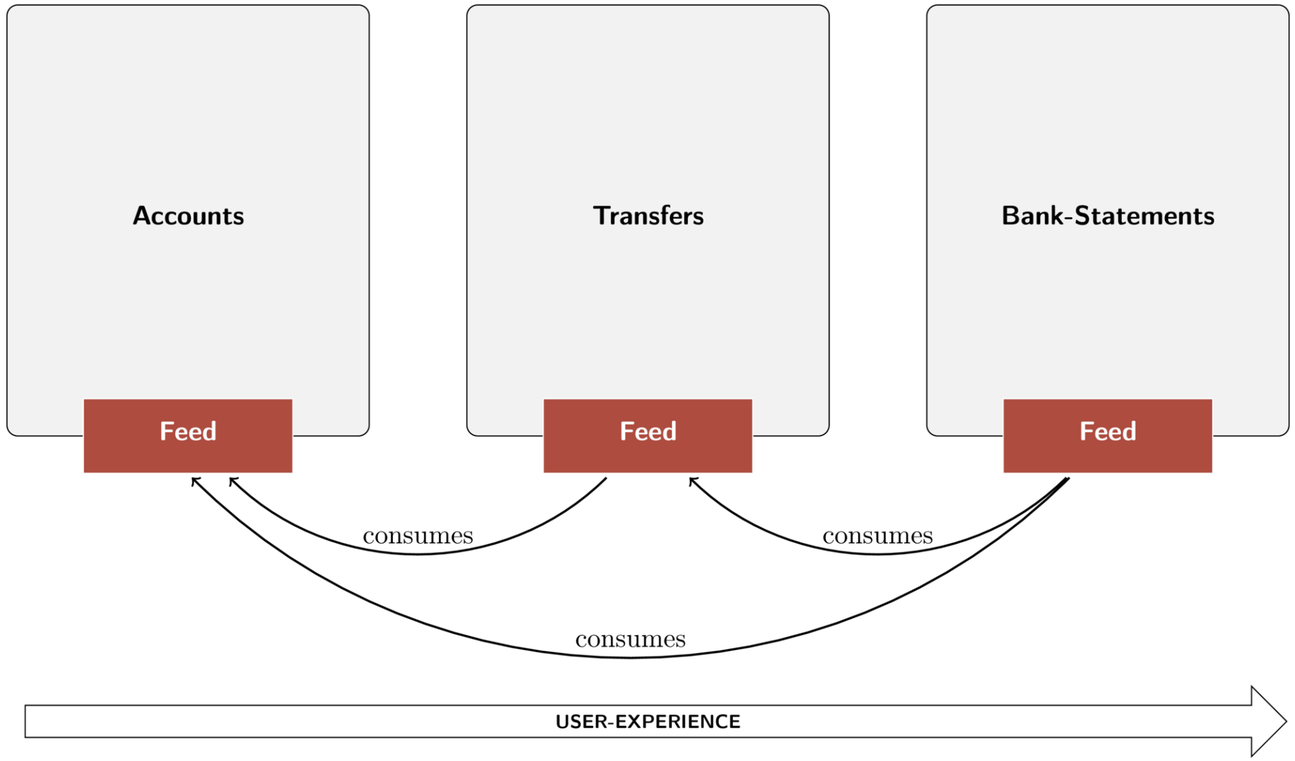
Discuss synchronicity vs asynchronicity.

- Why is synchronous communication dangerous?
- What is eventual consistency? What are the consequences for our applications?

Exercise: Data Integration #3

What would be a good macro architecture for Erlbank? Discuss alternatives and assess quality goals for each of them!

Erlbank: Feeds



Feed & Consumer Management

A **feed**

- offers the history of events, starting from a pointer
- is started with the application
- uses a database as the data source

Feed & Consumer Management

A **consumer**

- on startup pulls the whole history, memorizing the last pointer
- pulls periodically beginning from the memorized pointer
- persists events in database in own format

Feed & Consumer Management: Failure

If a **feed dies**

- consumer temporarily gets no answer. No problem due to asynchronicity
- it gets restarted and returns as before

Feed & Consumer Management: Failure

If a **consumer dies**, it

- pulls the whole history again
- in the worst case it stores events twice
- if last pointer is persisted it just continues to read

Exercise: What are the drawbacks?

Discuss the drawbacks that become visible during the integration. What are the benefits we achieved so far?

Drawbacks

- More code, more complexity
- Methods of communication need to be communicated between teams
- Complexity of integration

Benefits

- Loose coupling in code as well as development
- Clear boundaries between services can be established
- Latency and response times can also improve through locally optimized storage layout

Specification

- Methods of communication must always be specified
- Feeds, APIs or Database Schemas must be described for other teams, so that they can be used
- If this specification changes there must be coordination. E.g. versioning with downwards compatibility, notifications if a new version is available, ...

Integrating Legacy Systems

This requires an interface to the legacy system - integration via:

- direct calls
- messaging
- database
- UI

With all but UI, this requires translating the *data models*, for example via the anti-corruption layer model.

What is still missing?

Frontend integration!