Avaloq Client Meeting 06/12

Software Engineering Team D
6th December 2017

Progress Update

Progress at the first meeting

At our first meeting on the 15th November, we had the following:

- Basic Event Bus
    - Written in Rust
    - Events sent to Kafka and other connected clients
    - Events received from websocket connections
    - Kafka/Zookeeper instance running in Docker with a Docker Compose file
    - Full logging with different levels, including from underlying libraries
    - Full command line interface with arguments and subcommands
- Basic Demo Application
    - Static HTML page with no persistence
    - Communicated over websockets in Javascript
    - Demonstrated events being received, state being adjusted and events being sent

Progress since last meeting

Since our last meeting, three weeks ago, we’ve completed the following:

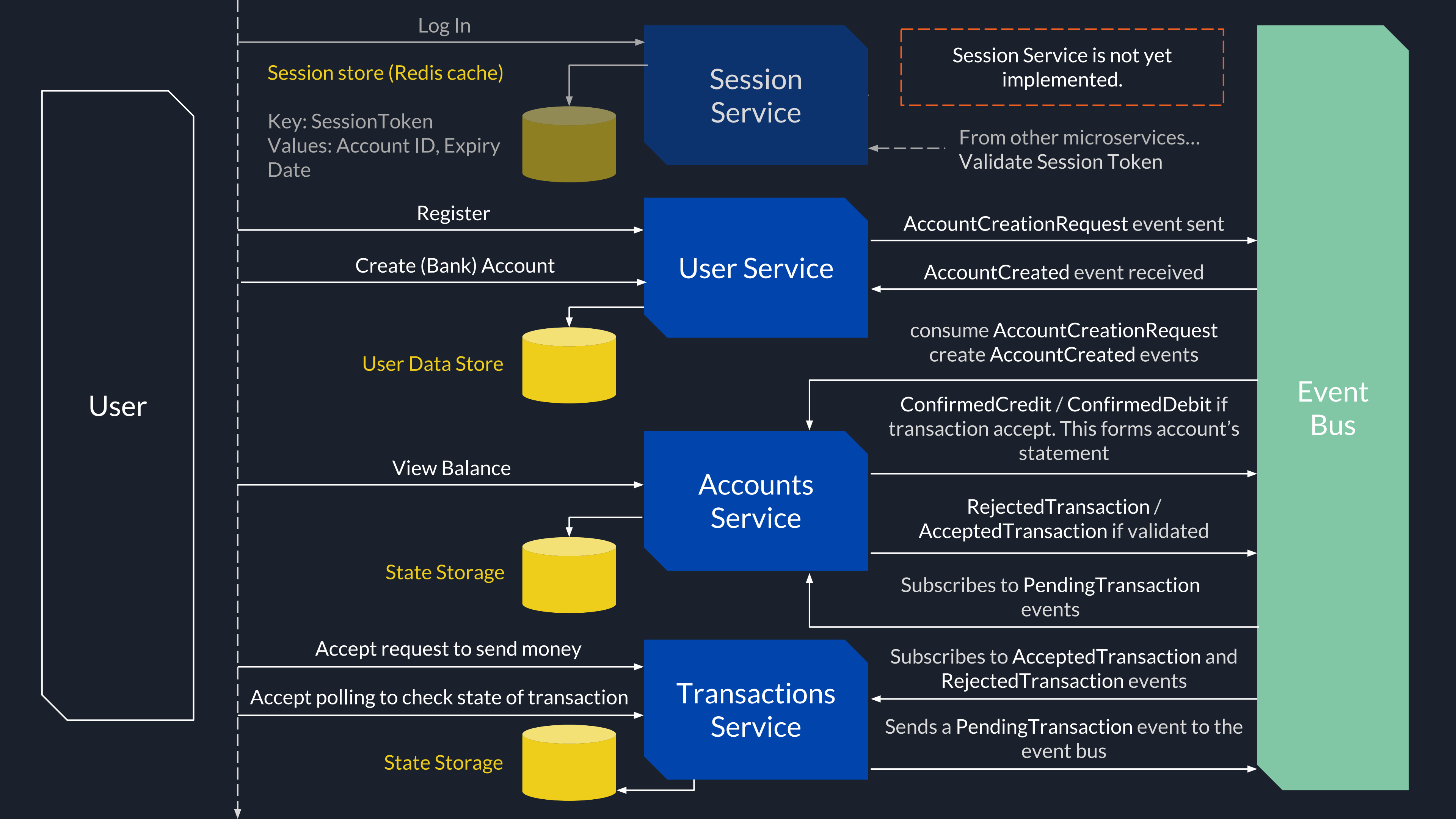
- Split into multiple repositories
- Improved Event Bus
  - Worked out a specification for messages and events and implemented this in the event bus
  - Improved handling of errors
  - Started work on producing receipts for messages
  - Unit testing of producer and consumer modules
  - CI pipeline for running tests and building docker image of the event bus
- Event Bus Persistence
  - Couchbase instance in Docker Compose configuration
  - Kafka Connect instance for persisting Kafka’s events
  - Groundwork for querying the event bus for previously sent events

Progress since last meeting

- Java Client Library
  - Provides a nice interface to the event bus and handles communication over websockets
  - Unit testing with 95% coverage
  - Testing, compilation and production of JAR in CI pipeline
- Transaction Service
  - Handles user requests to send money from one account to another
  - Creates PendingTransaction events and allows the user to query the status of their transactions
- Accounts Service
  - Processes PendingTransaction events , with outcome either Accepted or Rejected depending on the balance of the From account
  - Creates ConfirmedCredit and ConfirmedDebit events which form the user’s account statement
  - Keeps internal state tracking the accounts balances
  - Processes AccountCreationRequest events that are sent from the User Service and responds with AccountCreated events

Progress since last meeting

- User Service
    - Allows users to Register with a username and password
    - Allows users to create multiple accounts (as in money-holding things)
    - Currently no implementation for login / sessions
    - Stores user data in Postgres (passwords stored salted and hashed)
- Misc
    - Docker registry added and integrated with GitLab
    - User stories written for all client services
    - All services persist a local state (not a source of truth) built up from the events.


Log In
Session store (Redis cache)
Key: SessionToken
Values: Account ID, Expiry Date
Session Service
Session Service is not yet implemented.
From other microservices...
Validate Session Token
Register
Create (Bank) Account
User Service
AccountCreationRequest event sent
AccountCreated event received
User Data Store
consume AccountCreationRequest
create AccountCreated events
User
Event Bus
ConfirmedCredit / ConfirmedDebit if transaction accept. This forms account's statement
View Balance
Accounts Service
RejectedTransaction / AcceptedTransaction if validated
State Storage
Subscribes to PendingTransaction events
Accept request to send money
Accept polling to check state of transaction
Transactions Service
Subscribes to AcceptedTransaction and RejectedTransaction events
Sends a PendingTransaction event to the event bus
State Storage

Plans for the next meeting

We'd like to achieve the following before our next client meeting:

- Query functionality
  - Allowing clients to query the event bus for previously sent events
  - Builds upon the Couchbase work already completed
- Consistency/Ordering
  - Implement the consistency plan (we'll come onto this in few slides).
- Register messages
  - Allow clients to limit which event types they are interested in
- Login/Sessions on the client apps
  - Perhaps a basic UI if we have time

Plans for the next meeting

- Receipts
  - Initial event bus implementation is complete
  - Would need to integrate this into the client library

..and of course anything that you want prioritized/worked on.

Demo

Process

Agile, Sprints and Stuff

Since the last customer meeting we have implemented and experimented with a few different variations in our process, such as:

- Pair Programming / Mentored issues
  - Prevent knowledge silos from forming within the team
  - Ensure expertise are spread widely throughout the team (despite different backgrounds/experience)
  - Ensures help is more readily available if someone is struggling with their assigned tasks
- Change Management
  - We have outlined a change management process to follow
  - All changes via Merge Requests and only once they are reviewed by another team member and have passed CI
  - We have disabled pushing directly to master on the key code repositories

Agile, Sprints and Stuff

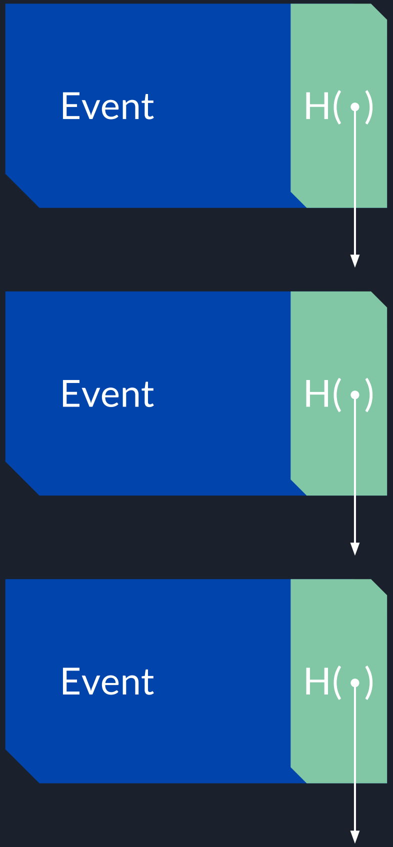
- Testing methods
  - Cucumber BDD - user stories -> scenarios -> Gherkin feature files
  - JUnit standard unit tests
  - Rust unit tests for eventbus itself
- Planning poker
  - Use of planning poker to estimate and prioritise issues
- Slack
  - Regular updates whenever we make changes, keep team in sync
  - Centralised point of contact for all
  - Use polly to decide on lunch location beforehand, that way we don’t waste time worrying where our energy for the afternoon is coming from ;P
- TeamUp Calendar
  - We have a shared calendar to track our meeting days / commitments that all have access to, on mobiles / shareable link.

Consistency / Ordering

Initial Idea: Blockchain-inspired ordering

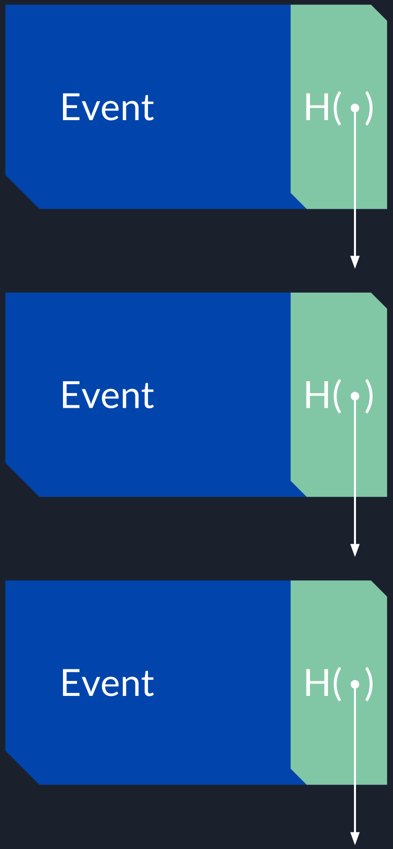
Our initial idea to enforce ordering was to include the hash of the previous event in each event, this would successfully enforce ordering, but had some downsides:

- Every client would need to receive all the events in order to have the most recent hash.
- It could be pretty slow.



Initial Idea: Blockchain-inspired ordering

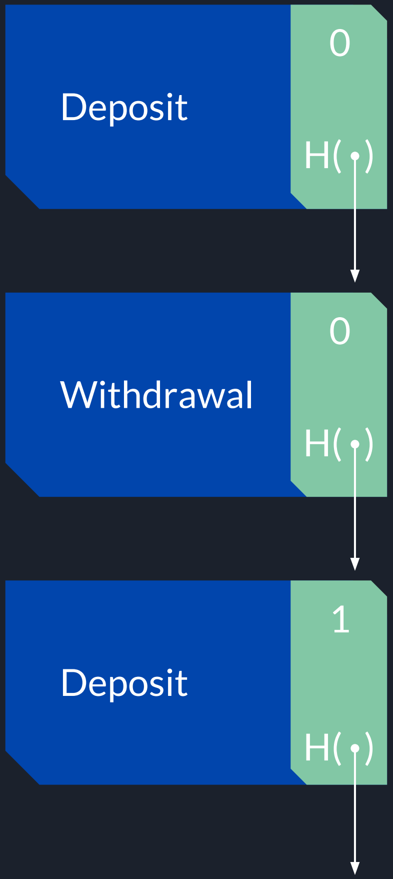
We decided that when a new event was rejected due to a hash being out of date, we would send the current top-of-chain hash in the rejection message. This allows hosts to avoid needing to listen to all events.



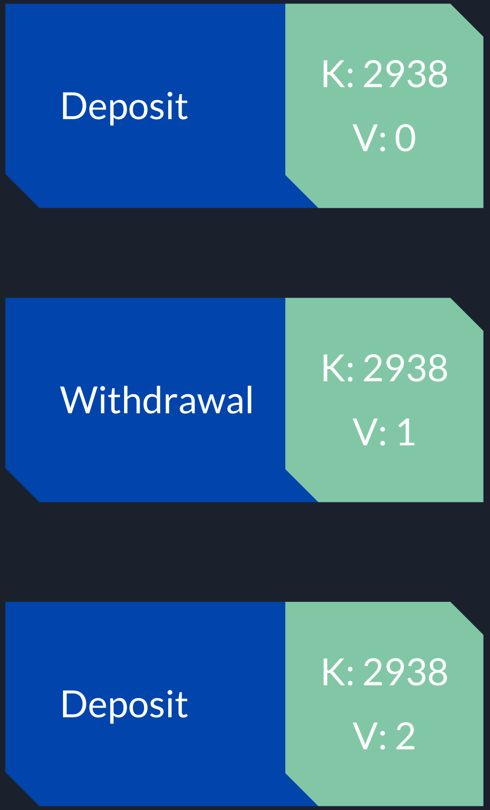
Improved Idea: Include a sequence number

We improved on the initial idea by sending a sequence number per event type. We do not send this with a rejection message. This ensures that a client must have read to the top of the chain for a given event type and have processed them all before sending new messages of that type. This approach had the following downsides:

- Probably pretty slow. If n messages are sent at once, then there are n! retries coming.



Current Idea:
Sequence Key/Value



Our current plan for maintaining consistency and ordering involves including a sequence key and value. Each event would include a key and a value, all events with the same key must include the incremented previous value.

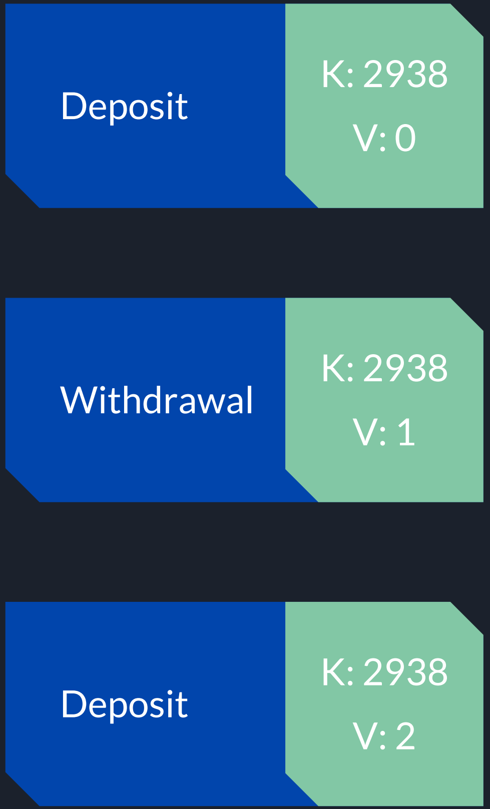
An sequence key might be a account number or anything - it’s arbitrary. This method allows us to maintain an ordering on events that could conflict - ie. deposits and payments on one account.

Current Idea:
Sequence Key/Value

We haven't thought of any downsides of this method yet.

We think it'll be quick enough as the only rejection of event should occur when they could modify the same balance (for example) while not slowing down everything to maintain a global ordering.

Implementation should be relatively simple too.



Any questions ?