Avaloq Client Meeting 21/02

Software Engineering Team D
21st February 2018

Our primary focus for this sprint was delivering a complete working system that hit all the initial requirements.

Iteration 5

We've completed a lot this month, we're going to be deep diving into some of the bigger things that landed this sprint in upcoming slides. Here are some smaller completed tasks:

- Consistency is fully implemented, working and tested.
  - This is a big deal but as it has been covered and discussed before so we've included it here.
  - Also includes persistence of the consistency between runs of the event bus.
- Population Script
  - Allows creation of lots of data for demonstrations and can test the entire system under load.
- Deposit and withdrawal endpoints implemented in accounts service
- More work on Dissertation
  - Now at 15 pages out of the 20 maximum.

User Interface and UI Backend

User Interface and UI Backend

A big focus this sprint was getting a working user interface so we can demonstrate an end-to-end working system. To achieve this, we've worked on two separate projects:

- UI Backend (or BfaF/Backend for a Frontend)
  - Acts as a gateway between the various microservices with their REST APIs and the web application itself.
  - Written in Python with Flask and now completed.
- UI
  - React application that speaks to the UI Backend.
  - Majority of functionality is in place - account overview; statements and transactions are included. Withdrawals and deposits nearing completion.
  - Work is progressing on the layout and style.

Multiple instances of services

Multiple instances of services

After the last meeting, we reviewed our current system to identify any potential issues relating to the horizontal scaling of our microservices. In order to implement this, we added something we've called *Sticky Round Robin*.

To avoid multiple services processing the same event at the same time, we initially introduced a round robin system, where events were distributed to one connected instance (rotating) of each type of service (transaction, user or accounts in our system).

This solution would work, but given that each instance would have its own internal state of what the outgoing consistency values are for each consistency key, a round robin solution would result in a lot of inconsistent messages as no one service gets all the events for a consistency key.

Multiple instances of services

To avoid issues with consistency, after we send an event with a given consistency key to a service in our round robin system, we send all future events with that consistency key to that service and only continue to use the round robin system for new consistency keys.

Each service type shares the internal datastore that it uses with all instances of that type. This allows any service to respond to queries over HTTP about any account; or if that information is required in the processing of an event.

We've implemented this in our services and working.

Recovery from service downtime

Recovery from service downtime

We can recover from all services of a given type being down by introducing a system of acknowledgements.

When a event is sent to a client, after all processing is complete, the client should send back an acknowledgement.

If a client crashes or is stopped, then any unacknowledged events for that client are sent to other services of that type (if available) or persisted until a service of that type connects.

This ensures that all events are processed. It is currently implemented and working.

Rebuilding and redelivery

Rebuilding and redelivery

When a client is started, it sends a query to the event bus with the timestamp of the last event that service type saw (zero if it is completely new).

This query is responded to with all of the events that have occured since that timestamp (that the client has registered for), this allows the client to rebuild its internal state for events that have happened since it was down.

This allows the services to rebuild their internal state if they are completely wiped. It is currently implemented and working.

Software Development Process

Software Development Process

Throughout this sprint, we have continued following an Agile software process that has worked for us in previous sprints:

- Continued use of mentored issues, found to be useful. Team members introduced to parts of the project through mentored issues now successfully making meaningful contributions without needing mentored issues.
- Retrospectives (including mentored retrospective with team mentors) being completed at the end of each sprint with SWOT[1] (Strengths, Weaknesses, Opportunities, Threats) analysis.
- All MRs are being reviewed by another team member - making heavy use of GitLab's review functionality to make comments around specific parts of the MR.

This slide is mostly for the University's benefit.

[1] See Silicon Valley SWOT Analysis: https://www.youtube.com/watch?v=XfB0g_JDIds

Moonshot:
Superclient

Disclaimer:

This is an addition to the project that we added because we saw some pain points in our existing codebase.

It was done on top of the normal work we'd planned for the sprint because it sounded like a lot of fun.

It wasn't necessary, is probably overkill and isn't one of your requirements - but it does make things easier for us.

Moonshot:
Superclient

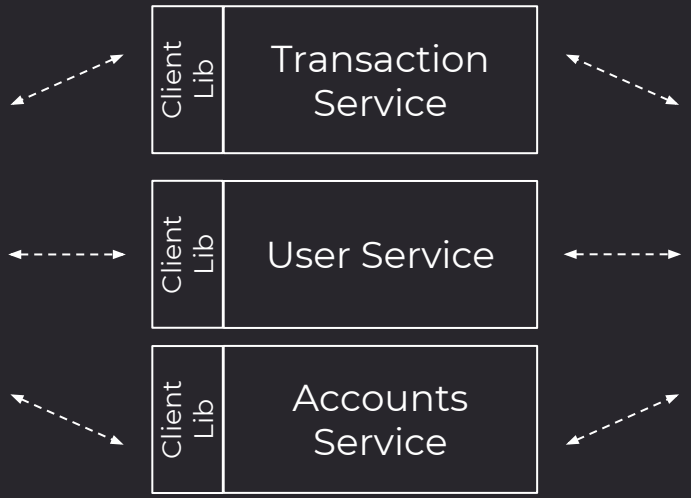
During development of some of the other additions this sprint, we noticed that a big bottleneck in our implementation work was our existing services (and the corresponding client library).

Adding new functionality that required changes or additions to the client library or services slowed down development significantly - delaying changes already completed in the event bus from being rolled out and thoroughly tested.

When considering limitations in our solutions to problems, it wasn't viable to implement small services to see how the interactions between them would work under our solutions to different problems - this was due to the heavyweight nature of our existing services and the time required to implement one.

Moonshot:
Superclient

Previous Architecture:



New Architecture:

Moonshot:
Superclient

The Superclient is a framework for creating microservices that handles all of the repetitive parts of making a service for the event bus - it goes further than the client library in Java could.

Previous Java Client Lib/Services:

- Client library handled communication with Event Bus.
- Each service handles web server and creating the REST API that is used for interacting with it.

Superclient (Rust with Lua Scripts):

- Loads business logic for each service from a Lua script that makes use of APIs for client-related functionality.
- Exposes APIs for subscribing to events, receipts, rebuilds.

Moonshot: **Superclient**

Previous Java Client Lib/Services:

- Each service handles connecting, creating schemas and managing PostgresSQL storage for that service.
- Client library ended up being ~1,800 lines and each service was between 650-850 lines, around 4000 lines of Java.

Superclient (Rust with Lua Scripts):

- Exposes APIs for persisting and querying data from Redis.
- Scripts are small and only handle business logic, each service is around 100-200 lines of Lua. Rust code is ~1,600 lines.
- All “protocol considerations” (consistency, correlation, multiple instances) handled by Rust code, Lua services don’t need to care.

Moonshot: **Superclient**

```
bus:add_route("/account/{id}", "GET", function(method, route, args, data)
    log:debug("received " .. route .. " request")

    -- Get the information we have stored about this account.
    local account = redis:get(PREFIX .. args.id)
    if account then
        -- Return some of the data.
        return HTTP_OK, { id = account.id, balance = account.balance }
    else
        -- Return an error if we do not have data.
        return HTTP_NOT_FOUND, { error = "could not find account with id: " ..
args.id }
    end
end)
```

Moonshot:
Superclient

```
-- Handle request for an account creation.
bus:add_event_listener("AccountCreationRequest", function(event_type, key,
correlation, data)
    log:debug("received " .. event_type .. " event")
    -- Get the next ID.
    local last_id = redis:get(ID_KEY)
    local next_id = last_id.id + 1
    redis:set(ID_KEY, { id = next_id })

    -- Create a new account and send the event out.
    create_account(next_id ,data.request_id, true)
end)
```

Moonshot:
Superclient

Advantages:

- Faster and easier iteration for us when working on it.
- Easier to experiment and try new things.
- Still familiar to the team as architecture is similar to event bus.

Disadvantages:

- Probably not practical for complex services that have *lots* of logic, can still write full services.
- Almost certainly overkill.

In general, we've found that the Superclient is a lot easier to maintain and allows us to iterate quicker - business logic is small and easy to understand; all the "protocol" code is simplified and shared between services. It is completed and working and will be used in the upcoming demo.

Demo

Our primary focus for the next sprint is polishing the entire system, working on some other requested features (reporting, etc.) and writing our dissertation.

Any questions ?