Phase 3 - Design decisions

In this document the design and implementation of Agent-Smith, a distributed storage network, based on the pithos architecture, will be discussed.

More specifically, the design and implementation of `phase-3`.

Phase goal

The goal of phase-3, was to improve the existing features implemented in Agent-Smith and prepare the application for real world application. High level features implemented in this phase include:

- Moved away from localhost testing
- Improved leader selection
- Improved Dynamic Grouping
- Basic Storage API for external services

Component/Feature implementations

The following sections will discuss each component/feature in more detail

Features implemented as part of `phase-3`:

- External Directory server
- Moved away from localhost testing
- Improved leader selection & re-election
- Implemented redundant Super-Peers
- Improved grouping mechanism
- Improved (DHT) Overlay storage
- UML Application diagrams
- Data layer improvements
- RESTful Storage API

Moved away from localhost testing

As part of any service implementation, one starts off testing locally, until the application reaches a point where the isolated testing environment no longer serves a constructive purpose, but rather as rose tinted glasses. Agent-Smith has reached this point of development, and it was therefore very important that we move away from localhost, towards a multi-host environment.

To achieve this the following changes needed to be made,

- Extract Directory server
  - Investigated cloud providers possibilities
  - Run Zookeeper instance on a single node Kubernetes cluster
- Update leader selection to support multiple groups (leader per group)
- Update group storage to dynamically look up leader address
- Store DHT port information per peer/super-peer within the directory server

This was as one can image not easy to achieve. As an initial step to achieve independence from localhost testing, it was decided to move the Directory server to an external host. This would allow for greater network scale and introduce real world network characteristics into our storage network such as latency and allow us to start navigating various other complexities of real world networks.

To start things off, we investigated the possibility of hosting the Directory server on a cloud provider such as AWS. Seeing as the approach was also aiming at spinning up a single node Kubernetes cluster, this proved to be quite tedious and expensive - the cloud provider approach was abandoned soon after.

After seeing the potential of Kubernetes as an orchestration tool, it was decided to implement a local single node Kubernetes (K8s) cluster to host a directory server. As things stand, a single Zookeeper instance is active within the cluster, in the future this can easily be expanded to add multiple pods to the network as this is fully supported by zookeeper and facilitated extremely well by K8s

In order to allow complete independence from localhost testing, it was required to introduce dynamic port selection/host binding per component (overlay-storage, group-storage, super-peer server, peer server). All components were updated accordingly and now support dynamic port selection.

Improved leader selection/re-election & Implemented redundant Super-Peers

In the past, leader selection was quite a big issue, as it was very flakey, and unreliable. Within the `Pithos` documentation, the concept of redundant super-peers are discussed. This means that in every group more than one super peer is created, but only one receives incoming requests from joining peers. In the event of a super-peer leaving the group, the next peer in line simply becomes the active super-peer, the group is then re-formed and not destroyed. In future development phases, when peers receives data from super peers, we'll need to think about how not to lose data or drop requests during this transition time.

Future Improvements:

- Move Health-Check to Super-Peer
- Peers not receiving group objects from super-peer
- Handling requests during transition phase
- Using object/peer ledgers

Improved grouping mechanism

Previously only one group could be handled. This was mainly due to only having one leader system wide. After implementing leader selection per group, it was easy to implement grouping properly. In `phase-3` of development, the groups are added sequentially, i.e. if the first group was *group-1* the subsequent group name will be *group-2* and so on. This can be improved in future development phases.

Future Improvements:

- Groups are currently limited to a fixed size of 3 peers per group.
- Group naming can be improved.

Improved Group storage

As part of `phase-3`, **Group-storage** was greatly improved. Previously, groups-storage was quite fallible in the sense that if a Peer or super-peer became unhealthy during the course of the networks up time, the group would not be able to recover from the faulty state. To improve this, a *Health check* was introduced. This health check is nothing more than a scheduled RPC, which checks the state of the gRPC group storage server of each client connection. If the state is not seen as `SERVING` i.e. (`NOT_SERVING`, `UNKNOWN`), the client will be removed from Group storage. This mechanism will ensure that unhealthy nodes stay out of the group.

Future Improvements:

- Move health-check to the Super-Peers
- Investigate RabbitMQ for message queues

Improved (DHT) Overlay storage

To ensure that each peer/super-peer (node) that connects to the same DHT storage and doesn't simply create an isolated DHT storage of it's own, we need a known host/port combination. To implement this, the Directory server was used as a way of making each node's DHT port known. This means that any node can connect to the overlay network, via any other node that is currently in the network.

The issue of knowing who the other nodes in the network are, was especially evident for super-peers, as they can't simply ask request a DHT bootstrap from any other node in their group, seeing as they are usually the first peers in the group. Some mechanism to know the names of groups was therefore required. In it's current state, Agent-Smith's groups are named sequentially i.e. *group-1*, *group-2* etc - if a group is therefore not the first group in the network it can always reach out to the previous group to find a bootstrap port. This can however be improved by creating a method to return all group names from the directory server, relieving us of the need for sequential group names.

We also moved away from binding to loopback address and binding to an available network interface of the host. This was quite challenging as we need to only bind to physical network interfaces and not the interfaces of VMs and containers. (this could also change to support multiple VM interfaces that don't not use a bridged adapters)

One other important thing to note is that TomP2P was not designed for java-8+n and on an older version of netty. We therefore need to ensure that the overlay network performs as expected. It could also come in handy to reach out to the maintainer of the project to do a review of the code.

Future Improvements:

- Create a method to return all group names from the directory server
- Ensure stability of overlay network

UML Application diagrams

As a nice to have, a UML diagram generator was used to generate UML diagrams for each class. This will definitely come in handy when writing the official thesis documentation.

Data layer improvements

Working with multiple data storage layers (overlay, local, group), increases the complexity of transferring objects between layers. Fields like `creation time` and `data values (byte arrays)` required tedious mapping code, to ensure that we keep the integrity of the data throughout the system, without accidentally corrupting the object data. `MapStruct` definitely came in handy here, which allowed us to automatically generate mappers for these objects, with the help of more detailed mapper classes of course.

A security improvement was also introduced by moving to prepared statements, instead of raw SQL string executions. This means that our API requests are no longer vulnerable to SQL injection.

Future Improvements:

- Investigate whether the `creation time` fields is required
- (If so) introduce Time Zones into internal models

REST Storage API

Finally, as part of moving towards a usable storage network, a simple storage **RESTful** API was created. This API supports simple **CRUD** *(Create, Read, Update, Delete)* operations, that invoke overlay-, local- and group-storage. In future development phases, the API could be implemented as a RPC API as-well, depending on the project requirements. Furthermore, we could extend the API with *path variables* for *read/write* mode.

Future Improvements:

- Investigate project requirements (RESTful/RPC API)
- Introduce Path variables for *read/write* modes

Technical dept for phase-3

Tech dept from `phase-2`:

- ~~Redundant Super peers~~
- ~~Dynamic group assignment~~
- ~~Zookeeper Leader Electors per group~~
- Dynamic group sizes
- Research RabbitMQ for **Main Controller** class
- Implement repair of objects
- Implement migration of peers
- Implement secure gRPC connections (enable TLS/SSL)

Tech dept from `phase-3`:

- Move Health-Check to Super-Peer
- Peers not receiving group objects from super-peer
- Handling requests during transition phase
- Using object/peer ledgers
- Improve group naming strategy
- Dynamic group sizes
- Investigate RabbitMQ for message queues
- Create a method to return all group names from the directory server
- Ensure stability of overlay network
- Investigate whether the `creation time` fields is required
- (If so) introduce Time Zones into internal models
- Investigate project requirements (RESTful/RPC API)

- Introduce Path variables for *read/write* modes

Phase-4 features

The following features should be implemented in `phase-4`, along with some tech-dept.

- Usage of Group/Object Ledgers
- Repair of objects
- Replication options
- Group geo-awareness