Phase 4 - 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-4`.

Phase goal

The goal of phase-4, 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 ping functionality to super-peer
- Implemented group-ledger
- Implemented object replication & repair
- Implemented group migration

Component/Feature implementations

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

Features implemented as part of `phase-4`:

- Moved ping functionality to super-peer
- Improved grouping mechanism
- Implemented group-ledger functionality
- Improved gRPC performance
- Added various test scenarios
- Improved Object TTL functionality
- Removed MongoDB functionality
- Migrated all components to unix time
- Implemented object replication, repair & quorum mechanism
- Implemented peer group-migration

Moved ping functionality to super-peer

Pings are scheduled tasks, where each peers selects another peer in a uniform fashion, it will execute an tcp ping by creating a socket to the peers group storage port. If the socket is created without failure, the peer is reachable. If the creation of the socket connection results in an exception 3 times in a row, the peer is seen as not reachable - The peer will then send an rpc to the group's super-peer (SP), containing the peer in question's ip and port. The peer will then execute 5 tcp "pings" to the peer, if the peer is reachable, then the connection issue is between the two peers, however if the peer "pings" fail, then the peer is indeed not reachable and should be removed from the group. This is done by sending an rpc to all peers within the group, to remove said peer.

Steps:

1. Scheduled task: select a peer out of the list of available peers

2. Execute tcp "ping" to peer
3. Ping succeeds or fails
    1. succeeds -> continue
    2. fails -> retry
    3. fails 3 times -> Elevate to SP
4. Sends rpc `pingPeer` command to SP
5. SP executes the same ping command to the peer in question
    1. succeeds -> continue, peer connection issue needs to be resolved by peer
    2. fails -> retry
    3. fails 5 times -> Peer is down, should be removed from group
6. SP sends `removePeer` rpc to all peers in the group
7. Peers remove broken peer from group
8. Object repair is executed

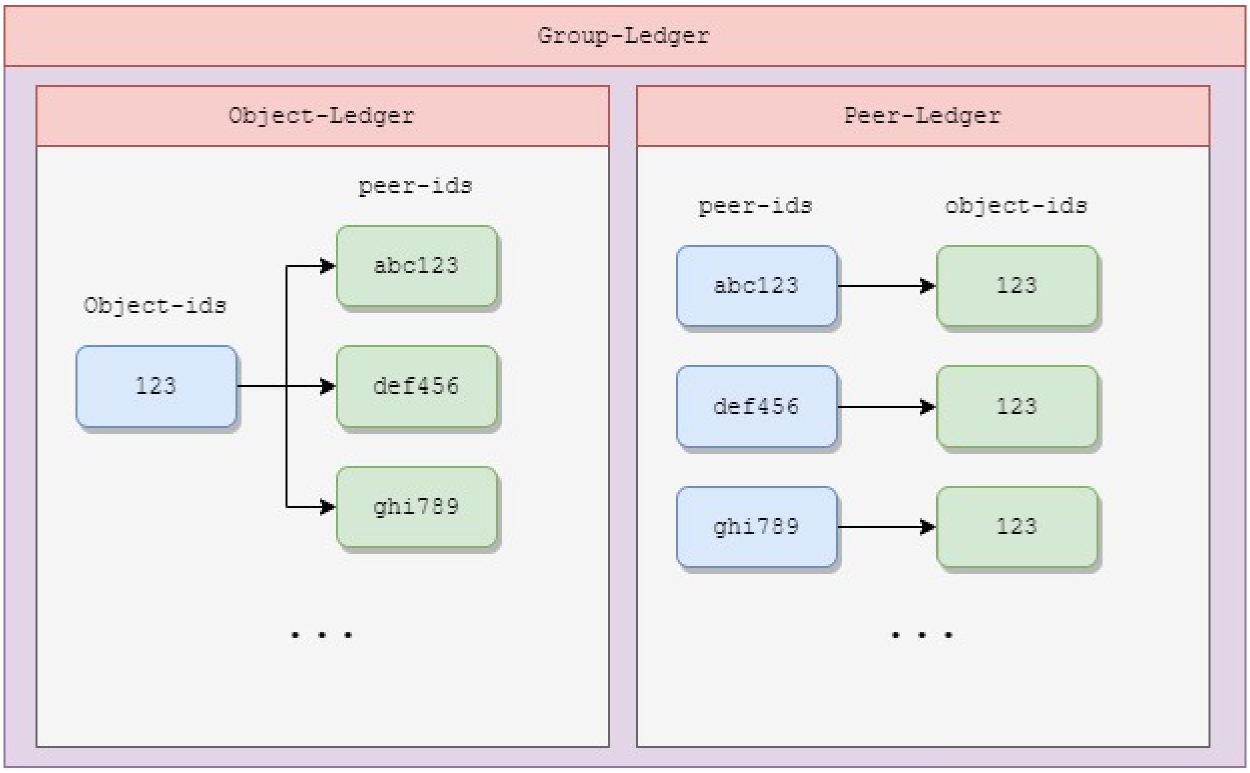
Improved grouping mechanism

Groups are now formed by checking the group's available spots, if there is a spot available according to the peer, then it becomes a `join-able` group.

Implemented group-ledger functionality

An important part of any distributed system, even more so for distributed storage systems is knowing where *what* is stored. To achieve this, the pithos architecture introduced the concept of a `group-ledger`. The group-ledger, keeps track of one thing, which peer stores which object. To simplify things, the group ledger provides inverse lookups.

In terms of implementation, the group-ledger can be described as an object consisting of two multi-maps with the signature `Map<Key, List<Values>>`. One map containing the peer-id as key with the list of object ID's as values and the other map, its exact inverse.



One implication with this approach is, that it results a lot of overhead network traffic for notifying other peers/super-peer that an object was added/removed/modified in a peer's database. One way to improve this, is to lessen the required calls. We achieved this by storing the objects TTL along with the value, this way we can simply clean up expired objects instead of having to ensure consistency between the ledger and the peer's database, thus for *deletes* no additional ledger requests are required.

Improvements points:

- Sharding
- Dynamically create the inverse of the map instead of storing two, nearly identical, maps
- Find a smarter way of synchronising group-ledger events
    - All writes are executed by super-peer
    - Peers update their view periodically

Added various test scenarios

To improve reliability of the various components that make up the storage network, many *missing* tests (unit & integration) were added. This allowed for various bug fixes and performance improvements.

Improved gRPC performance

Some investigation was done on using blocking gRPC clients vs future clients.

Test:

Create a client that does work for 30 seconds, calling the gRPC service functionality in a uniform fashion. Change from using a blocking client stub calls to a future client stub and measure the total amount of rpc requests processed.

Service used:
Group-storage service
Because the system is technically in a simulated state, a "working" time of `30* ms` for service operations was assumed.
(In the real word this could be more or less depending on network latency)

Results:

Blocking Stub:

```
gets: 427
puts: 425
updates: 447
notifyObject: 467
notifyRemovePeer: 424

Did 53.63333333333333 RPCs/s
```

Future Stub:

```
gets: 461
puts: 410
updates: 437
notifyObject: 463
notifyRemovePeer: 449

Did 56.46666666666667 RPCs/s
```

From the tests above, we see a slight performance improvement by using future stubs. It was therefore decided to use **future stubs** to perform gRPC request.

Migrated all components to unix time

The concept of time in a distributed system is very complex, especially when it comes to internal clock synchronization. We will not be discussing this in this section, and might even be out of scope of this implementation.

Here a different complex facet of *time* in distributed systems is tackled, namely conversion. Pre phase-4 each component (i.e. directory-server, DB, API, etc) required it's own implementation of *java* time. This made things **very** complicated, as it required a lot of mapping between different implementations. This opens the network up to a lot of conversion losses.

To avoid this, the concept of `unix` time was introduced, which in java can be represented as a `long` value. This in-turn greatly simplified object model conversions.

Improved Object TTL functionality

As mentioned in group-ledger functionality section, the concept of object TTLs were improved within the system.

The following improvements were made,

1. Scheduled clean-up query executed on the internal DB to remove any expired objects
    - Also cleans expired object in query results
2. Scheduled clean-up function to remove any expired references in the object ledger
    - Also cleans expired object in query results from the group-ledger

Removed MongoDB functionality

To keep the project within scope, the MongoDB components we're removed. If needed, they can be re-added in the future.

Implemented object replication, repair & quorum mechanism

Replication

> "Replication improves retrieval responsiveness, if multiple copies of an object may be requested in parallel and the object that arrives fastest is used. This is termed parallel retrieval in the Pithos design and is evaluated in Chapter 6."

Two storage modes exist within the network, `fast` and `safe`. This mode determines how many rpc results the system needs before it determines the operation result. In `safe` mode, the call is propagated to all peers within the group, for get requests a quorum mechanism is used to determine the result. In `fast` mode, the system only returns the first available result immediately. The initial read-write mode is set at startup and updated once connected to a SP.

Pre phase-4 storage requests were propagated to all peers within the group, effectively the network was always working in `safe` mode.

Repair

> "Repair further improves reliability by maintaining sufficient object replicas. When it is discovered that a peer has left the network, peers storing the objects of the peer that left can replicate those objects, thereby maintaining a sufficient number of replicas."

The repair mechanism was implemented by scheduling a task every `30 s`. The super-peer runs the repair task by checking it's group-ledger for objects that have less than the required replica count. The SP then checks the group-ledger for peers that don't store the object, creating a repair multi-map `Multimap<PeerClient, String>`, of peers & objectIds which they need to repair. Once the operation is completed, `repair` rpcs are sent to the peers in the repair multi-map.

The peer receives a list of objects, which it will iterate over, querying group-storage by the object (requiring repair) id. Once all object are repaired, it will send the repair result back to the super-peer.

Steps:

1. SP Scheduled task (30s): check group-ledger for objects that have less than the required replica count
2. Get a list of peers that don't store the object
3. Create multi-map of `Multimap<PeerClient, String>`
    - example

Peer-Client	Object requiring repair
client-1	["111", "222", "333"]
client-2	["222", "333"]
client-3	["111", "333"]

4. Sends rpc `repair` to peers in the map
5. Peer receives rpc, with list of objects
6. SP sends `removePeer` rpc to all peers in the group
7. Peers remove broken peer from group
    1. Iterate over the list
    2. Get Object from group-storage
    3. Store result (if any) in local storage

8. Send repair result back to SP

Quorum

> When security is a concern, multiple parallel retrieval requests can be performed. A quorum mechanism is then used at the receiving peer to compare the different received objects. The object that is returned by most of the peers is then considered to be the correct object.

The quorum mechanism merges group storage responses, merges the responses from group-storage clients and returns the `mode` result.

`quorum: [0 - 1]`
Parameter that can be set before application run,and determines how strick the application is on `get` requests.

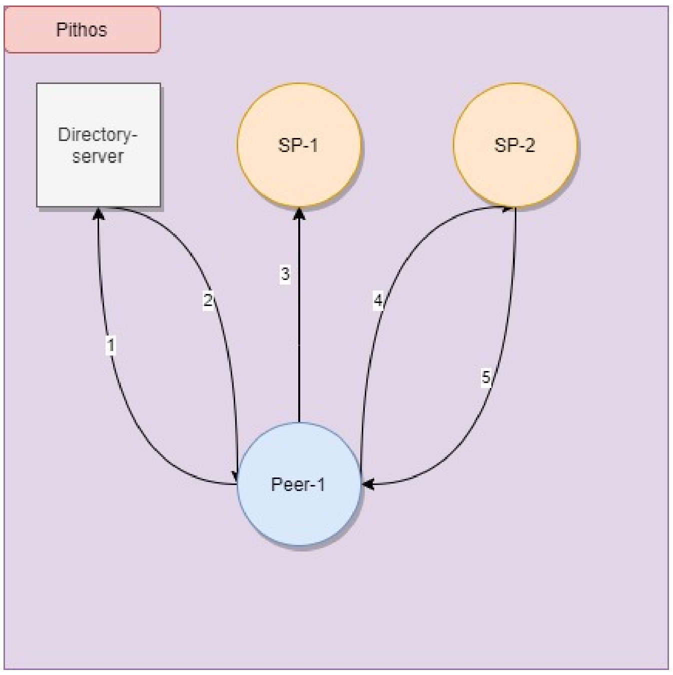
Example:

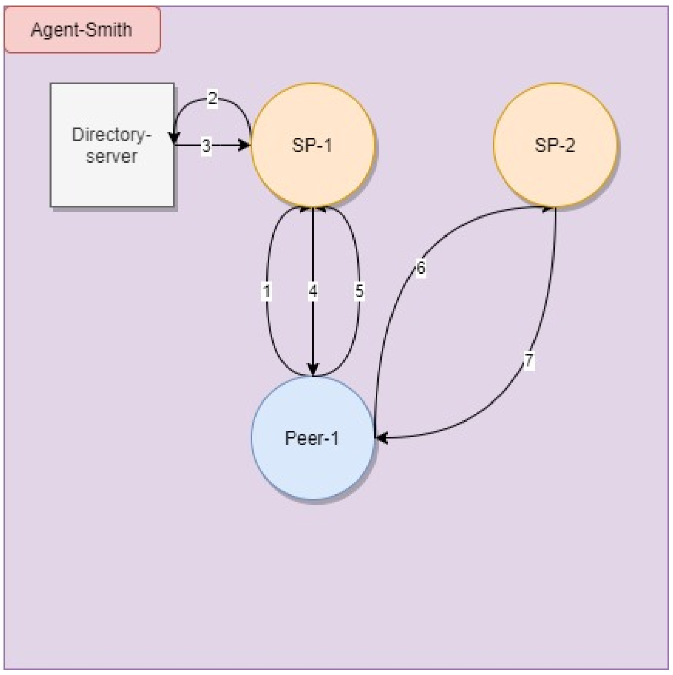
quorum	number of peers	required quorum
0.2	5	2
0.5	10	5
0.7	20	14

Implemented peer group-migration

Pithos group migration *(figure below)*, in it's current state, is not very efficient and could lead to a bottleneck as all position updates are sent to the directory server, which evaluates the position of the peer and sends a super peer id as ack, if the super-peer received in the ack is different from the super-peer the peer is connected to, it will initiate group migration. To make this functionality more efficient, it was *moved to the super-peer*.

This does not mean the SP won't get overloaded with migration requests, however, the chances of this happening is far lower than if we leave the migration mechanism in the directory server.





Pithos implementation:

1. Peer send position update to DS
2. DS responds with SP id
    - ID is the same as current connected SP - Do nothing
    - ID is different - initiate group migration
3. Sends rpc `leave` request to old SP
4. Sends rpc `join` request to new SP
5. Receives rpc `join result` from new SP

Agent-Smith implementation:

1. Peer send position update to SP
2. SP checks if the peer is still within the AOI
    - Inside AOI - respond with ACk
3. Outside of AOI
    - Asks directory server for neighbouring SPs
    - Chooses closest neighbouring SP
4. SP sends `group-name` and `new SP ip` to join
5. Sends rpc `leave` request to old SP
6. Sends rpc `join` request to new SP
7. Receives rpc `join result` from new SP

Technical dept for phase-5

Tech dept from `phase-4`:

- Dynamic group sizes
- Research RabbitMQ for **Main Controller** class
- Implement secure gRPC connections (enable TLS/SSL)

Phase-5 features

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

- Group geo-awareness
- Dynamic group sizes
- Implement secure gRPC connections (enable TLS/SSL)
    - Simple Certificate authority