Distributed store

Table of content

1. Overview

In this document, we describe software project, which we proposed for subject NPRG023 at Charles University in Prague. This document contains both the specification and detailed software project documentation.

Our team implements a distributed data store and a library to communicate with it, additionally; we provide a demo client application and tools for the functionality and performance testing. We implement the whole project in JAVA. The team consists of 4 students

- Tran Tuan Hiep
- Anton Khodos
- Andrea Turčanová
- Ladislav Maleček

And RNDr. Filip Zavoral, Ph.D, zavoral@ksi.mff.cuni.cz, will be a supervisor.

1.1. Brief specification

In this subsection, we give a brief overview of all components implemented in our software project. There are 4 major software components, that we implement:

1. Distributed data store
    - The main component of the whole project.
    - Runs on a cluster of server nodes.
    - Multiple clientssaves their in the store.
2. Client java library for communication with the store.
    - Users can easily integrate this library into their own codebase and start communicating with the data store.
3. Demo client application
    - Command line application
    - Serve as an example of the integration of the client library
    - We, also, used it for both functionality and performance testing.
4. Pcap analyzer
    - Application, that analyzes the network traffic from pcap file with sniffed packets.
    - We will use it for analyzing the throughput and the latency of the data store.

In addition, we create a core library, which we use for the implementation of all 4 listed software applications.

1.1.1. Distributed store

The distributed data store runs on a cluster of nodes. Clients use this cluster to store their data in different data models.

When the store is running, the clients issue read and write data request to the store, which processes their requests. The store guarantees to the clients, that loss of one of its cluster node does not result in the loss of the client's data. Additionally, new server nodes can be added to the cluster when the store is already running.

We presume that the store is never deployed on large scale and all store nodes are deployed in a single data center and connected with high quality network; therefore, the goal of this project is not to implement large scale distributed store deployed on multiple data centers all over the word; instead, we focus on developing a store deployed on a small scale within a single data center. Additionally, the store is always deployed on a linux servers.

1.1.1.1. Data models

The clients saves their data in multiple data models within the store. And the store, itself, can support many data models.

By default, the store supports two data models for its clients, the key-value map and a list. However, the distributed store is easily extensible with new data models. When it is running, the clients are storing each of their data sets in instances of these data models. For example, if the clients wants to store three different datasets, they can choose to store the first data set as an instance of a list and the other two as an instance of key-value map.

Additionally, the store supports querying on top of the clients data, but we do not implement the querying globally in the store. Instead, we provide a mechanism for an easy integration of querying with the data models. I.e. each data models can implement the querying themself. We choose this approach as opposed to the global mechanism for querying because the querying can be specialized and highly optimized for its data model.

Next, the store persists all clients data on the disk; however, the store keeps a portion of the clients data inside its main memory for a better performance. The distributed store has a reasonable performance, when the client data on every node fit in its main memory.

Finally, the store ensures a replication of clients data on multiple nodes.

1.1.2. Client library

Java library that implements communication with the data store. Clients can easily plug this library into their application, and start issuing read and write request to the data store cluster.

To start working with a distributed data store, clients must establish a connection between them and the store cluster by using this client library. Once the connection is established, the clients can send their request to the store using this library.

However, the library does not have an API to directly manipulate with the actual data model instances stored in the store; instead, the clients use so called *data model communicators*.

1.1.2.1. Data model communicators

A *data model communicator* allows the client to work with a data model instance stored in the cluster. The API and functionality this *communicator* corresponds to its data model. For example, the *list data model communicator* is specific for the *list data model*. Clients can add an item, remove an item, or access an item of an *list data model* instance through the *list data model communicator*. In contrast to the *list data model communicator*, a *key-value data model communicator* allows the client to add a new mapping between key and a value, remove an existing mapping or access a value with a given key of an existing instance of *key-value data model*.

Underneath, these *communicators* use the client library to communicate with the store. I.e. the client library is responsible for transferring a client requests to the cluster over the network layer, while the *data model communicators* are responsible for creating these requests.

In addition, the client library provides an easy mechanism to integrate with new *data model communicators*, just as a distributed data store to integrate new data models. The client library, also, has an API to create new *data model instances* and to remove the existing ones.

Rapid Addition, further, requires that the implementation of this client library is GC free.

We now recapitulate all relevant definitions regarding the *data model*:

- *Data model*
    - Model that defines how client data are stored inside the distributed data store.
- *Data model instance*
    - Actual instance of the data model holding clients data inside the running data store.
- *Data model communicator*
    - Component that is used within the client application to work with the *data model instances*.

In addition, we add the following definition:

- *Data model plugin*
    - Data model plugins contains all components that are necessary for the clients to start using with a data model.

- Both *data model* and *data model communicators* are part of a data model plugin.

We will be referencing these definitions in next sections.

1.1.3. Demo client application

Command line application, which tests the store functionality and displays the integration of the client library. We use it mainly for extensive functionality and performance testing. This application simulates a client in our integration and performance tests.

The client application runs multiple testing scenarios and has an easy mechanism for integrating new testing scenarios. We provide at least 3 scenarios as a part of the project, two for functionality testing and one for performance testing.

1.1.4. Pcap analyzer

We use the client application, described above, to benchmark the store, but the client application does not store any relevant data about the performance evaluation itself. We instead capture the network packets into a pcap file. Pcap analyzer then processes the captured packet and outputs the latency and throughput report of the distributed store. We use hardware timestamp from the network card for performance analysis.

The described approach is more precise than the approach with timestamping and performance application inside the testing application. Additionally, it does not intervene with the distributed store and the client applications. We benchmark the store in the performance lab of Rapid Addition.

1.1.5. Core library

The core library contains the common functionality for each component. All our software components reference this core library. We create the library to avoid code duplication.

The library contains utilities for config parsing, multithreaded programming, and network communication and so on. Next to that, the core library wraps 3rd party libraries and adapts them to our use case.

1.1.6. Technology

Rapid Addition requires us to implement the whole project in Java 8 and to use gradle as our build system.

In addition, we use git as our versioning system, and we plan to deploy jenkins as our CI system. The jenkins will be deployed in cloud; we, also, want to run our integration tests in cloud.

1.1.7. Summary

In the following sections, we give a detailed description and design rationale of each component developed by us. At first, we describe the cluster architecture, which corresponds to the high level point of view, and then we focus on a more low level stuff, such as internals of the distributed store or core library content.

2. Distributed store cluster

In this section, we describe the cluster architecture. At first, we describe a typical deployment of the distributed store and a interaction between the distributed store cluster and its client. Then, we describe the cluster architecture with one example of a typical use case scenario. Finally, we describe the communication among the distributed cluster nodes and between them and the clients.

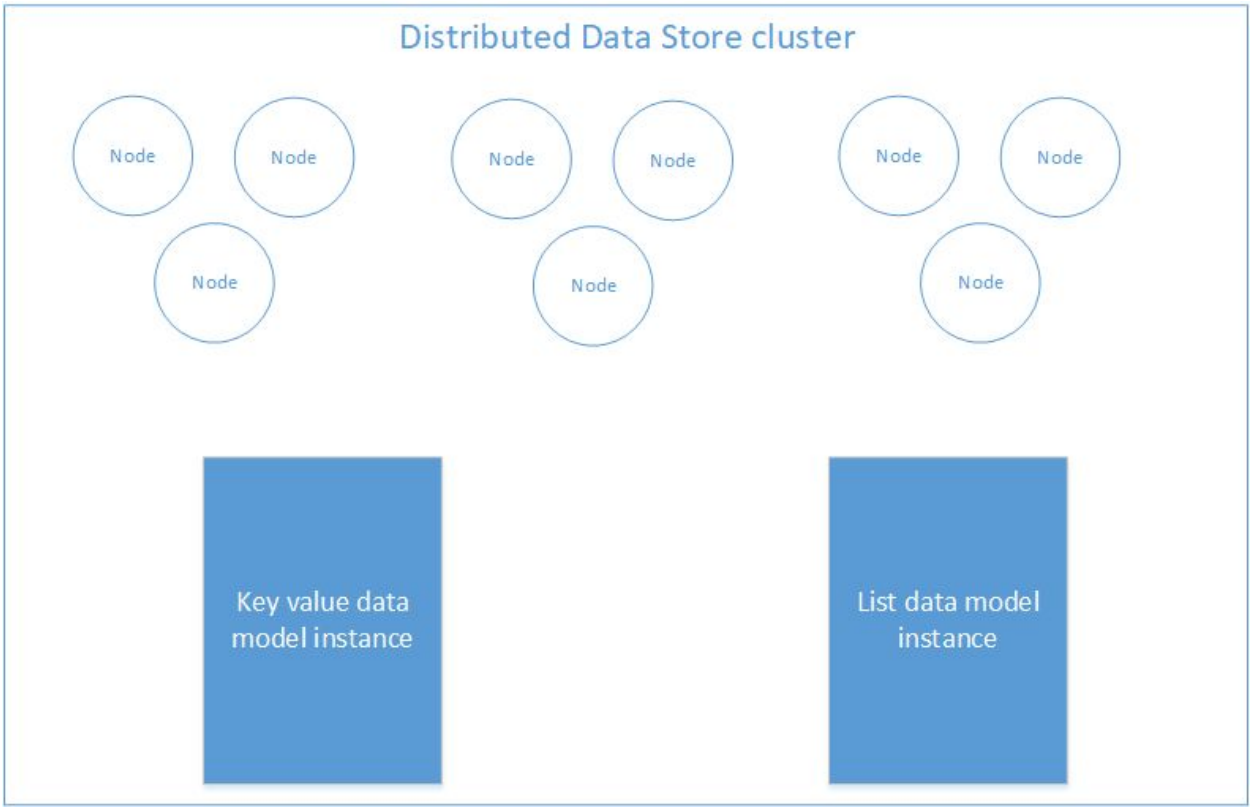
2.1. Typical deployment

As we already stated, the distributed store usually runs on a small number of nodes. Also, small number of clients uses this store.

The store allows the client to save their data in multiple data models, such as list or a key-value map. When the store is deployed, it contains instances of these data models, which holds the actual client datasets.

Let us assume, that we deployed the distributed store with the support of the two basic data models, the *list* and the *key-value map*. The clients use the list for storing a continuous sequence of their data items and the key-value map to store a mapping between one type of their data items and another type of their data items.

For example, if the distributed store contains one instance of a list and one instance of key-value map then each client can access both instances. In case of the list, each client can issue requests to insert a new item, remove an item, or access an item to this particular instance of the *list data model*. In case of a *key-value map*, they can issue a request to insert a new mapping between a key and a value, delete an existing mapping, or access a value for a given key. Once, the distributed store processes the client request; it returns to the client a response with the request result. Following diagram describes the example situation.



2.2. Cluster architecture

The distributed store guarantees to its clients, that their data are not lost when one of distributed store nodes fails. To ensure this, the store replicates each *data model instance* to multiple nodes of the store cluster. Now, we describe how we replicate the data model instances within the cluster.

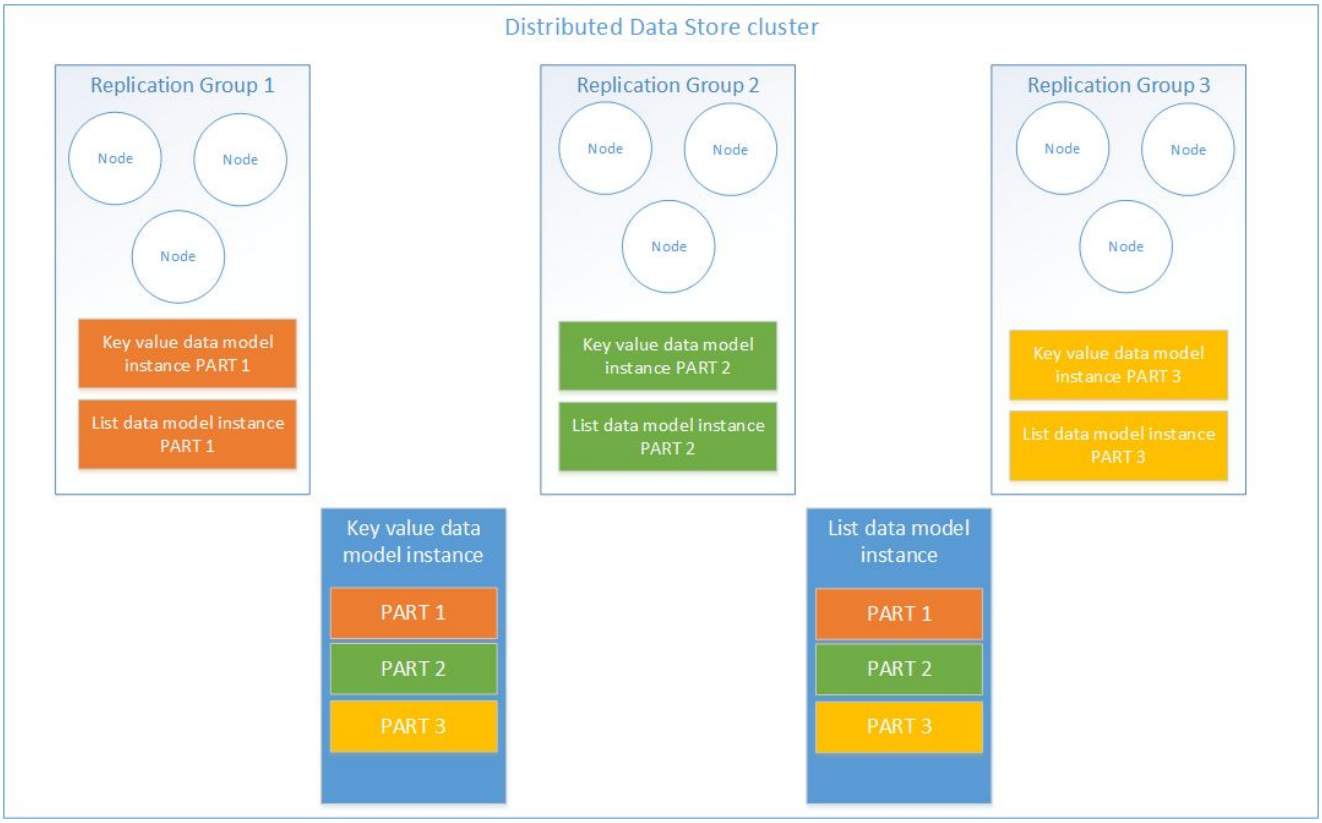
2.2.1. Replication groups

We define a replication group within the data store cluster as a group of nodes storing the identical client's data. In addition, following conditions must hold:

1. Each cluster node must belong to one and only one replication group within the data store cluster.
2. Each *replication group* in the store cluster must have at least three nodes.
3. Two different *replication groups* cannot store the same client's data.

We split the whole data store cluster into these replication groups, and each replication group must hold different set of client's data. Furthemore, we divide each data model instances into as many parts as there are replication groups within the store cluster, and we store the individual parts in different replication groups.

For instance, let us consider the example of typical deployment, mentioned in the previous subsection, where we had 9 nodes inside a cluster and 2 instances of data models, an instance of a key-value map and an instance of the list. Then, if we have 3 replication groups, and each replication group 3 nodes, then we divide both instances into 3 different parts and store individual parts in different replication groups. The following diagram, just like the one in previous subsection, pictures this situation.



We summarize that the replication of data works as follows: Clients are storing their data in multiple instances of data model instances; we split each data model instance into n parts, where n is a number of replication groups within the data store cluster, and then we store each individual part of the data model instance into a different replication group. These individual parts of a data model instance are replicated by all nodes within the replication group.

2.2.2. Design rationale

The described approach to data replication is a good compromise between a performance, fault tolerance, and complexity of an implementation.

2.2.2.1. Replication groups

If we did not use the replication groups, we would have to choose between two alternative approaches. Either we would replicate each data model instance to every node within the data

store cluster, or each data item of the data model instance would be replicated into different set of the cluster nodes.

In the first alternative approach, the data cluster can cope with upto n - 1 node failures, where n is the size the cluster because we have n copies of the data stored in each node. On the hand, this approach affects performance due to the fact, which we need to replicate data to a larger number of nodes. In addition to replicating all data on every node of the data store cluster, this approach, also, increases the requirements on memory and disk space of the individual nodes within the cluster. By applying our approach, the data store cluster can cope with the loss of at least one node per replication group. While this approach lacks the fault tolerance of the first approach, it has better performance because each data item is replicated only to nodes within a replication group, which has smaller number of nodes than the whole cluster.

The second alternative approach for a replication is too complex for an implementation and offers us no clear benefit over the chosen replication mechanism because we would have to be able for each client data item within the data model instances compute the nodes the item was replicated.

We, also, split each data model instance and store individual part in different replication groups, which allows us to distribute the content of each data model instance among all replication groups. If we did not split the content, each data model instance would be stored in a single replication group. Consequently, if all nodes in the replication group failed, the content of data model instance is lost. In comparison, when we split the data model instance content to all replication groups, we lose only the part of the data model instance, which was in the replication group. Additionally, the execution of query operations on the whole data model instance is load balanced among all replication groups. If we did not split the content, the query operation could negatively affect performance of all nodes within the replication group storing the whole data model instance.

2.2.3. Routing of client requests

Each client request, except for the query request, must be routed to a single replication group. The *data model communicator* decides to which *replication group* the client library routes the request.

In this subsection, we have described the replication mechanism of client data in the distributed data store. We have also provided a rationale behind our replication mechanism. Since all nodes in a replication group store the same data, we need to describe how to keep the replicated parts data model instances in consistent state within each replication group. This problem is called a consensus problem, and it is the topic of the next subsection.

2.2.4. Consensus

Consensus is a problem, where multiple nodes must agree on values. In our context, nodes within each replication groups must agree on the current state of replicated part of each data model instance. For example, if we have an instance of key-value map stored in the data store cluster; we need make sure that a operation requested by the client is replicated by all nodes of the replication group which processes the client request, and that each node in the replication group must process the routed client requests in some form of ordering. The form of ordering depends on the level of a consistency.

2.2.4.1. Consistency types

There exist multiple types of consistency models, but eventual and strong consistency are mainly applied in the context of distributed databases. If we simplify, then eventual consistency guarantees, that each read operation from client eventually returns the newest written value. On the other hand, strong consistency guarantees, that read operation returns to the client always the newest written value. I.e. in the eventual consistency, the read operation does not always return the most up to date result, while in the strong consistency it does.

We choose to implement a strong consistency in this project because the eventual consistency adds an additional burden on the clients to handle the situation when the read request does not return the newest version. This effect can lead to unexpected state of client applications. Instead, by choosing the strong consistency, we make the interaction with the data store for the client transparent as possible without any weird side effects.

2.2.4.2. Consensus algorithms for strong consistency

There are many algorithms to implement a strong consistency among replicas. The two most widely used are Paxos [1] and Raft [2]. We choose to implement Raft to ensure the strong consistency because is relatively easy to understand and implement. In comparison to Paxos, which is known to be notoriously difficult to understand and even more to implement.

For this project, we implement our own version of Raft because the existing open source implementations in Java are not gc free; therefore, they do not meet the performance requirements of the project. Next subsection tries to give a very brief overview of the algorithm.

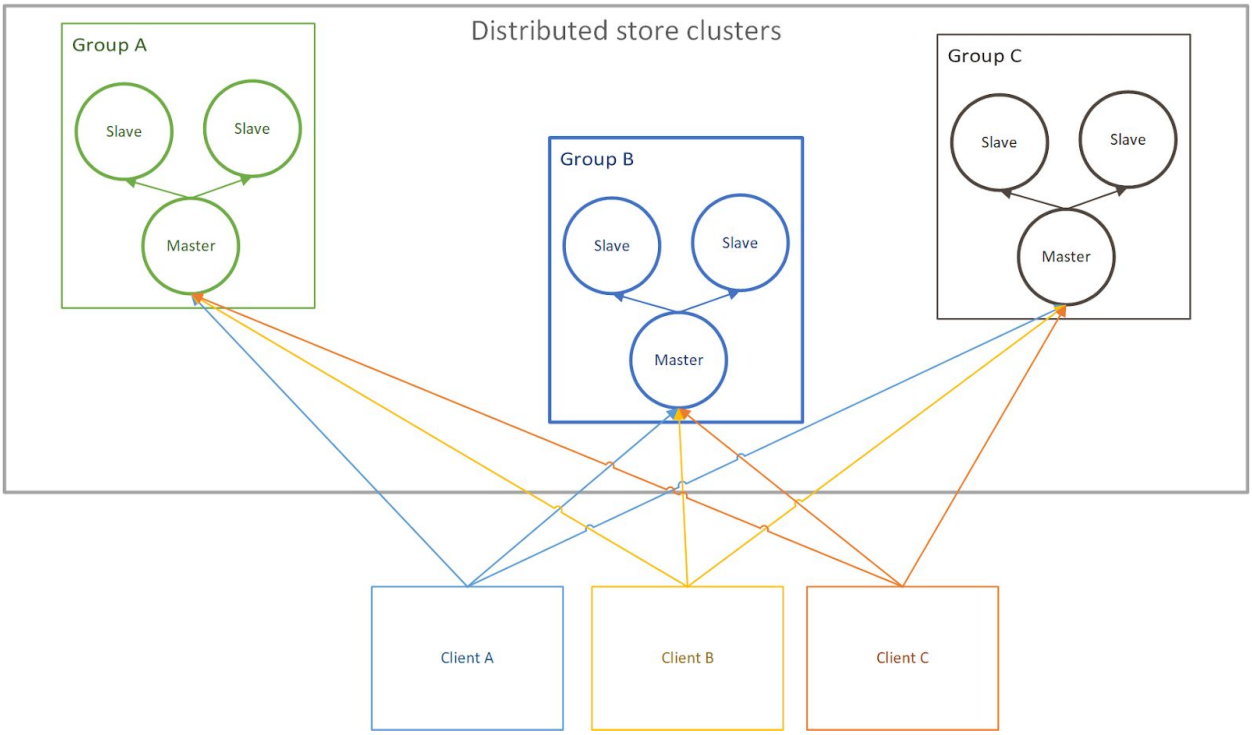
2.2.4.3. Raft

Raft is a consensus algorithm based on replicating of a state machine in each node within one group. It achieves this by replicating by a log of operation on the state machine in each node. Once raft replicates an operation in the log on a majority of nodes, the state machines on each node execute it. The algorithm uses master/slaves replication scheme for the state machine log, i.e. there is always one master and slaves, which always replicate the log of operation from their master. Additionally, raft contains a leader election mechanism to cope with the loss of master,

and it guarantees strong consistency among the replicas, i.e. each state machine sees the operations in the log in the same order. In addition, the log is persisted on the disk to ensure the fault tolerance.

In our context, each replication group has a single master and two or more slaves. The state of the state machine is the content of a stored data structures. An operation on that state machine is a single write request from the client. Therefore, the replicated log contains write requests from the clients. Since the read request does not change the state of the replicated state machine, we do not need to replicate the request to slaves; instead, we directly process it on the master.

The diagram, below, pictures a clusters of a 9 nodes within the distributed store cluster splitted into 3 replication groups and 3 clients issuing requests to the distributed store. When a client issues a request, we choose a single group, which process the request, and then we send request to master of that group. If the client request is a read, the master processes immediately the reads the required data and returns them to the client. On the other hand, if the client issues the write request, master replicates the write request to the slaves, and once the majority of slaves confirms that they have received the write operation, then the master executes the write and answers the client with a success.



2.3. Communication

In this subsection, we analyze how the distributed data store nodes communicates with each other and how the clients communicates with the data store cluster. At first, we define

communication channels, that are used; then we define how the communication channels are established. Finally, we discuss the implementation, which technology and a protocol the store and the clients uses for communication.

2.3.1. Communication channels

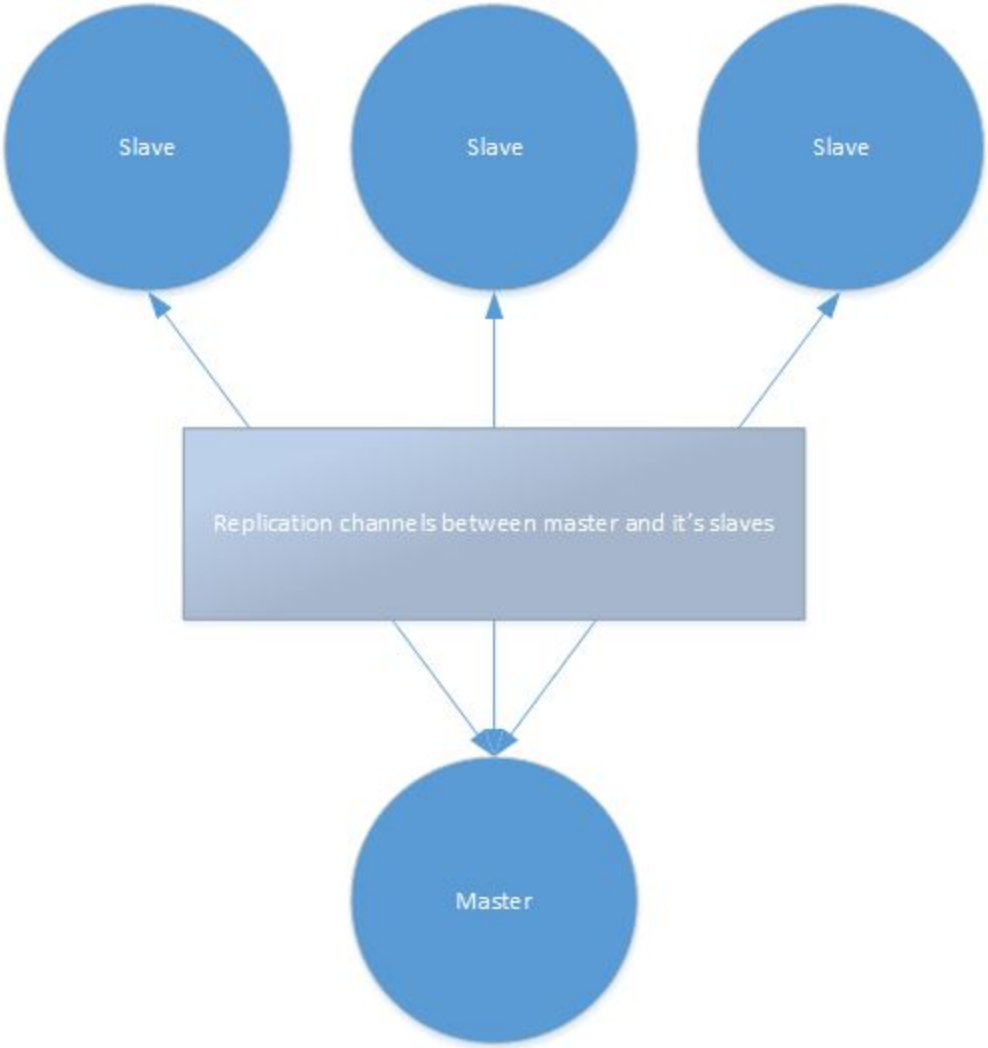
There are 3 types of communication channels in our project. Each of these channels is used to transfer different information.

1. Client ↔ Data store communication channels
    - Bidirectional unicast channel used for a communication between a client and masters of replication groups within the data store cluster.
    - Clients sends requests to the store and receives responses from the masters through this channel.
    - Client opens this type channel to all the master nodes of each replication groups.
    - When a replication group elects a new master, the client must close the old channel to the old master and open a new channel to the newly elected channel.
2. Raft communication channel within a single replication group
    - Bidirectional unicast channel used inside each replication group for a communication between the master and slaves.
    - This channel is used to transfer Raft specific messages to implement the consensus.
3. Client ↔ replication group master used for client initialization and cluster updates
    - Bidirectional multicast channel used for initializing the Client Replication group masters channels.
    - Clients used this channel to send a request to all replication group masters to get all necessary information that it needs to establish all channels to masters of each replication group.
    - Additionally, cluster sends Information about newly elected masters of a replication group, new node being added to a replication group, etc. flows through this channel to all clients.

We have defined 3 types of a communication channels. Next, we described their purpose and the parties that uses them for communicating with each other. In the next subsection, we define how they are established and their interaction.

2.3.2. Communication channels within the replication group

This channel is used mainly to transfer RAFT specific messages. This channel is initialized a when the distributed store starts. The following diagram depicts the communication channel inside a single replication group.

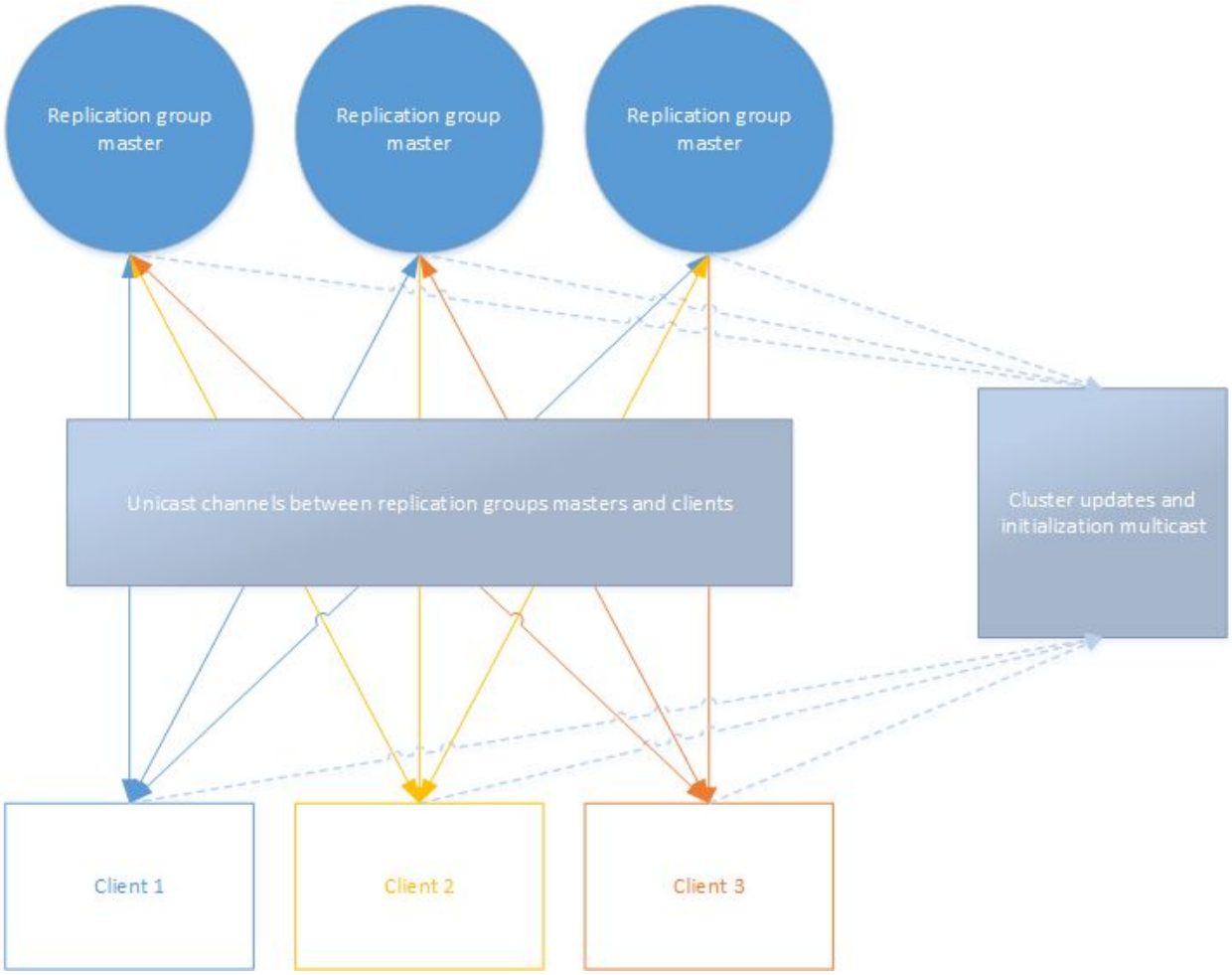


2.3.3. Communicating channels between the client and the store cluster

The clients communicate only with the masters of the replication groups since they cannot get any relevant information from the slaves which they cannot get from the masters.

When a client wants to connect to the master nodes, it uses one channel of *type 3* [2.3.1]. to get all the necessary information to initialize separate channels of *type 1* to each master node.

The following diagram depicts the communication channels between the replication master nodes and the clients.



2.3.4. Technology

We use the aeron [3] middleware to implement all our communication. Aeron provides a one direction unicast and a multicast channels to communicate between different parties. We build all the communication channels on top of these aeron channels[3.3.].

Aeron is a reliable protocol build on top of the udp. We decided to implement all of our communication on top of Aeron, because it is more performant than the other existing open source middlewares, such as amqp, activemq, zeromq and etc. Aeron also provides us with monitoring tools. This make the implementation of our communication on top of Aeron easier than building it on top of the raw TCP or UDP.

2.3.5. Message format

We use our own binary format for the exchanged messages in the distributed data store. We decided design our own binary format because using a text based format, such as xml or json, have a negative impact on performance of the distributed data store. On the other hand, we do

not need a complex binary format, such as bson, because the exchanged messages have a simple structure and using 3rd party binary does not pay off for a such a simple messages.

In this section about cluster architecture, we have described in detail how the distributed store interacts with the clients, how the nodes of the data store cluster interact with each other, how we replicate data and keep them in the consistent state, and finally how all components communicate with each other. The focus of this section was to describe to the the readers how the distributed data store works at the cluster level. Next, we focus on each individual parts of the project. We specify their functionality and focus on low-level details.

3. Core library

The core library implements functionality which is used all over our codebase. The library contains various number of utilities for configuration parsing, multithreading, network communication, and etc.

The two most important parts of this core library are library for DI injection build on top of Guice and our own reactive library.

3.1. Guice and dependency injection

3.1.1. Guice

Guice [4] is an open source dependency injection (DI) framework upon which each of our software applications relies on. In the core library, we implement thin layer around this framework.

By leveraging this layer, we build each of our software applications, the demo client application, the distributed store, and the pcap analyzer, as a plugin based applications. The DI framework, also, helps with the depencies management because it reduces the need for a manual hardwiring implementations to its interfaces when we build the object hierarchy inside our software. We choose Guice as the DI framework since we are most familiar with it.

Guice works on a so-called Hollywood principle: “Don’t call us, we call you”. In the software engineering terms, the clients of Guice delegate the responsibility for providing its dependencies to Guice.

For example, when we have a class Foo and that uses an interface IBar with an implementation Bar; we need to manually pass instance of Bar to Foo in Foo’s constructor. However, if we use the Guice, we bind the interface IBar to the implementation Bar using guice framework, and then guice construct the instance Foo and passes Bar to Foo as IBar itself. We do not need to provide the depency of IBar and Bar to Foo; instead, guice provides it for us.

The code snippet below demonstrate the basic usage of Guice on the previous example.

```
2
3  import com.google.inject.Module;
4  import com.google.inject.Injector;
5  import com.google.inject.AbstractModule;
6  import com.google.inject.Guice;
7  import com.google.inject.Inject;
8
9  public class Program {
10
11     interface  IBar { }
12
13     class Bar implements IBar { }
14
15     class Foo {
16         IBar bar;
17
18         @Inject
19         Foo(IBar bar) {
20             // the bar has a type Bar,
21             // guice delivers the binded implementation
22             this.bar = bar;
23         }
24     }
25
26     static class PluginModule extends AbstractModule implements Module {
27
28         @Override
29         protected void configure() {
30             bind(Foo.class);
31
32             bind(IBar.class).to(Bar.class);
33         }
34     }
35
36
37     public static void main(String[] args) {
38         Injector injector = Guice.createInjector(new PluginModule());
39         Foo foo = injector.getInstance(Foo.class);
40     }
41 }
42
```

The basic building block of Guice is a Module, an interface that contains our bindings. In the code snippet above, the class PluginModule implements an interface Module from Guice. Inside the configure method of PluginModule, we have two bindings, one which binds Foo and other which binds the interface IBar to the implementation Bar.

In the main method, we create an Injector, also Guice class, and we use the Injector to create an instance of Foo. The Injector creates an instance of Foo for us and injects the Bar into Foo constructor as IBar.

3.1.2. Plugin based architecture with Guice

In the previous subsection, we have describe how guice and DI frameworks generally works, the basic principles, and the Guice API. Now, we focus on the DI layer that our core library provides.

We build our DI layer on top of Guice and java ServiceLoader [6]. We use the ServiceLoader to load all plugins for each application.

At first, we describe what a plugin is, and how we integrate a plugin into existing application. Then, we focus how our applications loads the plugin using the infrastructure from the Core library.

3.1.2.1. Application plugins

Application plugin is a jar containing an implementation of an interface IPlugin. It is necessary to put the plugin on a classpath of our applications so that it can be integrated with the application.

The plugin creators must also bind their IPlugin using the ServiceLoader to the interface IPlugin. We use the ServiceLoader, before Guice, to load all implementations of IPlugin in all jars on the classpath of our applications. The code snippet below contains the interface IPlugin.

```
2
3  import com.google.inject.Module;
4
5  public interface IPlugin {
6
7      /**
8       * Returns the plugin name.
9       * @return name of the plugin.
10      */
11     String getName();
12
13     /**
14      * Returns plugin module which contains the binding
15      * of the plugin.
16      *
17      * @return module with plugin bindings
18      */
19     Module getModule();
20 }
```

As we can see, the IPlugin has two methods getName and getModule. The method getModule returns a Guice Module, like PluginModule in the previous code snippet, the returned Module, contains all the guice bindings that the plugin uses.

3.1.2.2. Loading plugins inside applications

In this subsection, we describe infrastructure provided by the core library for loading plugins inside our plugin-based applications.

At first, we describe two main components for the plugin loading, a class Application and an interface IApplicationService:

1. Class Application
    - Class that manages the lifetime of our applications.
    - Loads all plugins at the start of application.
    - Creates the Guice Injector from all plugin modules.
2. Interface IApplicationService
    - Service, that starts when its application starts and stops at the end of our application.
    - Class Application manages these services.

At the start of all our applications, the core library class Application loads all implementations of IPlugin using the ServiceLoader. Once, we have the instances of IPlugin implementations, we create the guice Injector from modules returned by method getModule on IPlugin. We then create instances of IApplicationService with the guice Injector. The concrete implementations of the interface IApplicationService are using the binding defines in the modules.

We implement the plugin loading and integration in two phases. In the first phase, we load all implementations of the interface IPlugin with a ServiceLoader, and then in the seconds phase, we create the guice Injector from all loaded plugins. Once the injector is build, guice injector creates for as implementation of the interface IApplicationService, and we start the application. We use this approach two phase approach because Guice lacks the capability to load all plugins from the jars on a application path which the ServiceLoader have. On the other hand, the ServiceLoader is a very simple utility that lacks many features of Guice.

We have described how the Guice, the DI framework, works, its benefits and how it help us to build a common infrastructure inside for pluggable application in our core library. Next, we focus on the Core library support for multithreading.

3.2. Core Reactive Stream library

The core library second important part is reactive library, which we implement ourselves. This library is a gc-free and allows us to define a handling of asynchronous and synchronous sequences of events. We will further on refer to the reactive library as *core.reactive* library. The

API and functionality of this *core.reactive* library is heavily inspired by both the ReactiveX [7] library and its java version [8].

The *core.reactive* library allows us to define handling of both synchronous and asynchronous sequences of events in a concise way. For example, we can use this *core.reactive* library to define handling of requests from network inside a single node of our data store cluster, where the network requests are basically asynchronous sequence of events. Or, we can also define a handling of deserialized objects from a file, such as pcap packets from a pcap, using this library where sequence of deserialized objects from file is synchronous.

We rely on this library; every time we need to implement a processing of asynchronous or synchronous events, such as handling client requests, receiving data from networks, processing packets from a pcap file and so on.

The *core.reactive* library, in essence, implements an extension of the observer and publish/subscribe pattern with functional operators over the sequences of observed or published events. We decided to adopt these reactive principles because our codebase as a result is clearer, more readable and flexible in comparison to if writing it in a more traditional imperative way.

We create own version instead of using the java reactivex library due to the performance considerations such as gc, mainly created by the boxing and existing inefficiencies of java generics. Additionally, we do not intend to implement the whole library; instead, we only need a small portion of the reactivex library functionality.

Finally, we adopt the *Reactive Stream*s api [9]. We do this because the *Reactive Streams* api is a part of the standard java library from Java 9 and will become in the future a base for reactive programming in Java. Our library can then potentially interop with other java libraries that also adopt this *Reactive Stream* api.

3.2.1. Reactive stream api

There are four interfaces in the *Reactive Stream* api, *publisher, subscriber, subscription*, and processor. We only give a very brief description of each; more details are in documentation of the reactive streams.

- *Publisher*
    - *Publisher* emits items to the subscribers.
    - Before *subscriber* starts receiving data, *Publisher* passes a *subscription* from to the *subscriber*.
    - One *publisher* can have more than one subscribers.
- *Subscriber*
    - *Subscriber* receives items from the publisher

    - *Subscriber*, always, receives data from only one publisher.
    - *Subscriber* uses the *subscription* from the publisher, to control how many items it receives or to stop receiving the data.
- *Subscription*
    - As we already discussed, subscriber requests items from the *publisher* through the *subscription.*
- *Processor*
    - Processor is both a *Publisher* and a *Subscriber*.

3.2.2. Flow

Flow is the basic concept in *core.reactive* library and it builds upon the *Reactive Stream* api. Each *flow* consists of one publisher, which emits events, multiple operators on the emitted events and one subscriber, consuming the events. Operators can for example filter our are transform the events before they are passed to the subscriber.

```
13
14          Flow.fromSource(dataFromNetwork) // dataFromNetwork is the flow publisher
15              .filter(this::isClientRequest) // filter operator
16              .map(this::toClientRequest) // map operator
17              .subscribe(clientRequestProcessor); // the flow subscriber
18
```

// code snippet jako ukazka

Flows are easily composable with one another, because one *flow* can be a *publisher* of a other *flow*. In addition, *flows* are always single threaded. To compose flows running on different threads, we need to use a *FlowConnector*. This component becomes a *subscriber* on the flow we are transferring data from and a *publisher* on a flow we are receiving the data.

3.2.2.1. Transferring data between flows on different threads and back-pressure

As we already discussed, the *FlowConnector* is responsible for transferring events between flows which run on different threads. We defined the flow as a single threaded component and created the *FlowConnector* concept to take advantage of the *Reactive Stream* api.

The main problem of transferring data between flows on different threads or, generally, in a producer consumer scenario is what to do when the consumer cannot keep up with the publisher. In our case, the consumer is the flow receiving the data and producer is the flow publishing the data. The mechanism that the *Reactive Stream* api provides is called propagating back-pressure [10]. The *FlowConnector* propagates the back-pressure from the consumer flow to the producer flow.

The *core.reactive* library provides two implementation of the *FlowConnector*. Both use lockfree queues. The queues are, of course, used to transfer data to the consumer thread;

moreover, we also used to signal to detect the back-pressure from the consumer. When the consumer thread cannot keep up with the producer thread, the queue eventually becomes full. The implementations of *FlowConnectors* propagates back-pressure once their queues become full.

The first implementation uses lock free queue backed by an byte array. To transfer an event to another thread, this implementation always serialize the event from the publishing flow into a byte array, then transfers this byte array using the queue to the producer thread. The producer thread then deserialize the byte array back to the event and passes it to the receiving flow.

```
27
28          ByteRingBuffer<ClientRequest> byteRingBuffer =
29                  new ByteRingBuffer<>(ClientRequest.serializer());
30
31          Flow.fromSource(dataFromNetwork) // read data on producer thread
32                  .map(this::toClientRequest)
33                  .subscribeConnector(byteRingBuffer) // transfer to the consumer thread
34                  .poll() // poll from the ring buffer in the consumer thread
35                  .subscribe(clientRequestProcessor); // process requests on the consumer thread
36
```

The second implementation uses lock free queue backed by an preallocated array of events. This implementation always copy the event from the publishing flow to it's preallocated array. Then, the queue transfers the copied event to the consumer threads which passes that event to the receiving flow.

```
15
16          PreallocatedRingBuffer<ClientRequest> preallocatedRingBuffer =
17                  new PreallocatedRingBuffer<ClientRequest>(
18                          ClientRequest.copyFunction());
19
20          Flow.fromSource(dataFromNetwork) // read data on consumer thread
21                  .map(this::toClientRequest)
22                  .subscribeConnector(preallocatedRingBuffer) // transfer to producer thread
23                  .poll() // poll from the ring buffer in the producer thread
24                  .subscribe(clientRequestProcessor); // process requests on consumer thread
25
```

In this subsection, we introduced the *core.reactive* library. We use this library to define handling of events, such as requests from network. In addition, this library also deals with passing events from one thread to another. Most importantly, the *core.reactive* has a concise and easy to adopt API which boosts our productivity.

3.3. Aeron

As we previously stated, Aeron [3] is middleware, which we use to implement our communication; it provides a unidirectional unicast and multicast.

Aeron is a reliable protocol build on top of the udp. We decided to implement all of our communication on top of Aeron, because it is more performant than the other existing open source middlewares, such as amqp, activemq, zeromq and etc. Aeron also provides us with monitoring tools. This make the implementation of our communication on top of Aeron easier than building it on top of the raw TCP or UDP.

The core library contains two classes for communication.

- *AeronSender*
    - Class responsible for sending data to the other side.
- *AeronReceiver*
    - Class that receives the data from the AeronSender, i.e. other side.

Both *AeronSender* and *AeronReceiver* are compatible with the *reactive.core* library and can be used in the *Flows*. *AeronSender* can be a *subscriber* in a flow and *AeronReceiver* can be a *publisher*.

3.4. Other utilities

The *core* library also contains number of other utilities, such as config parsing, additional support for multithreading, and so on. We usually tend to rely on third party libraries to implement these features but the core library either wraps these libraries, or it exposes them to the library users.

3.4.1. Third-party libraries

In this subsection, we provide the list of libraries which we rely on.

- Guava
    - Number of utilities which the Java standard library misses.
    - https://github.com/google/guava
- Agrona
    - Collection of high performance data structures and various utilities
    - We, mainly, use the the byte ring buffer to implement one of the *FlowConnectors* in *core.reactive library*.
    - https://github.com/real-logic/agrona
- Disruptor
    - Ring buffer used for passing events from one thread to another.
    - We use this library to implement the other *FlowConnector*.
    - https://lmax-exchange.github.io/disruptor/
- Carrot HPPC
    - Collection of high performance data structures.
    - This library contains hashmaps and list that do not rely on Java generics; as a consequence, they don't generate garbage.

    - https://github.com/carrotsearch/hppc
- Java thread affinity
    - Library, for pinning threads to specific cpu cores.
    - The thread pinning boosts the performance of the whole platform.
    - https://github.com/OpenHFT/Java-Thread-Affinity
- Pcap4j
    - We use this library in the *PcapAnalyzer* to help us parse the pcap packets.
    - https://github.com/kaitoy/pcap4j
- JUNIT
    - Java framework for writing unit tests.
    - https://junit.org/junit4/
- Mockito
    - Mocking library.
    - https://github.com/mockito/mockito

3.5. Summary

In this section, we have describe the most important parts of our *core* library, mainly guice and the *core.reactive* library. We heavily rely on this core library to implement the whole project.

4. Data model plugin and it's integration

In the previous capitol, we defined the concept *Data model plugin* [1.1.2.1] which contains all necessary components that are necessary to integrate so that the clients can start using the *Data model* inside the data store. I.e. it must contain both the *Data model* itself and the *Data model communicator*. We introduce a corresponding interface in this section; furthermore, we show how to create a new *Data model plugin* and integrate with both the *Data store* and *client library* using Guice.

4.1. Data model plugin interface

Plugin creators must implement a java interface *IDataModelPlugin* so that they can integrate the *data model plugin*. The following code snippet shows the interface. This interface can change during the development; however, the basic concept is the same. *IDataModelPlugin* has two methods; the first one returns the factory for *IDataModel* and the seconds one returns a factory for *IDataModelCommunicator*. *IDataModel* corresponds to the *Data model* and *IDataModelCommunicator* corresponds to the *Data model communicator*.

```
5
6  public interface IDataModelPlugin {
7
8      IFactory<IDataModel> getModelFactory();
9
10     ITaggedFactory<? extends IDataModelCommunicator> getCommunicatorFactory();
11 }
```

4.2. Data model plugin integration

To integrate a new data model with the data store and the client library, we must create a new *Application plugin* [3.1.2.1]. Inside that *application plugin* guice [4] module, we bind the corresponding implementation of *IDataModelPlugin* to its interface.

```
27
28     static class PluginModule extends AbstractModule implements Module {
29
30         @Override
31         protected void configure() {
32
33             bind(IDataModelPlugin.class).to(NewDataModelPlugin.class);
34
35         }
36     }
37
```

Finally, we need to copy the jar which contains the *application plugin* onto the classpath of the data store and the client application using the client library. Both applications loads the

concrete implementation of *IDataModelPlugin* and use the appropriate factory from the plugin. I.e. the data store uses the model factory and the client library uses the communicator factory. The loading of the concrete implementation of *IDataModelPlugin* is implemented with Guice.

4.3. Default data model plugins

As we already discuss, the store offers two data model by default, the list data model and the key-value data model and we provide two corresponding implementations of *IDataModelPlugin*, *ListDataModelPlugin* and *KeyValueDataModelPlugin*, as a part of our core library.

4.3.1. List data model plugin

The list data model allows the clients to store sequences of data items. The *ListDataModelPlugin* contains a factory for creating instances of the *ListDataModel* and *ListDataModelCommunicator*. Both of these classes have a generic type parameters which corresponds to the type of the stored data item.

4.3.2. Key-value data model plugin

The key-value data model allows the client to store mappings from a key to a value. The *KeyValueDataModelPlugin*, just like the *ListDataModelPlugin*, contains factory for creating of both the *KeyValueDataModel* and *KeyValueDataModelCommunicator*. Similarly to the list data model, these classes are also generics but they have two generic type parameters, one for the type of key and one for the type of a value.

4.3.3. Integration of the querying

None of the default data models provides support for the querying. However, the data plugin implementers can extend from the default data model plugins and implement the querying in their extensions. Moreover, the plugin implementers can extend from a default data model plugin with a specific generic type, and then they can implement the querying for the specific data type.

For example, if we want to store class *Foo* in the *list data model* and add support for querying of Foo, we create a new class *FooListDataModelPlugin* which contains factories for the *FooListDataModel* and *FooListDataModelCommunicator* where the *FooListDataModel* extends the *ListDataModel*<Foo> and similarly the *FooListDataModelCommunicator* extends the *ListDataModelCommunicator<Foo>*. We can then implement *Foo* specific querying inside these extended specialized classes.

We choose the described approach of adding querying support to the default plugins instead of having a single querying mechanism in the default data model plugins, which the users can slightly extend because it is more flexible. Additionally, sometimes the client do not want to

query on their data stored in the default plugins so there is no reason to add unnecessary querying mechanism.

4.4. Design rationale

Integrating new *data model plugin* is straightforward because it fully leverages the DI framework *Guice*. Moreover, this approach does not require any change on the application code side, i.e. data store and client library code. If we did not use Guice [4], we would have to change the application side to integrate new data model plugin, which is cumbersome and error prone solution. By using *Guice*, we only need to put the *application plugin jar* on the classpath of both applications.

The described approach is very flexible. In addition, it makes the integration of a new data model plugin very easy from the point of the plugin implementers because they only need to implement few interfaces and then copy a jar and us, the application creators, since we do not have to change the existing application code to integrate a new data model.

5. Store application

We have described how the data store works on the cluster level. In this section, we analyze how the store application running on a single cluster node works. We split the cluster nodes running the store application into a replication groups[2.2], and inside each group, we implement the consensus algorithm RAFT [2][2.2.4.3.]

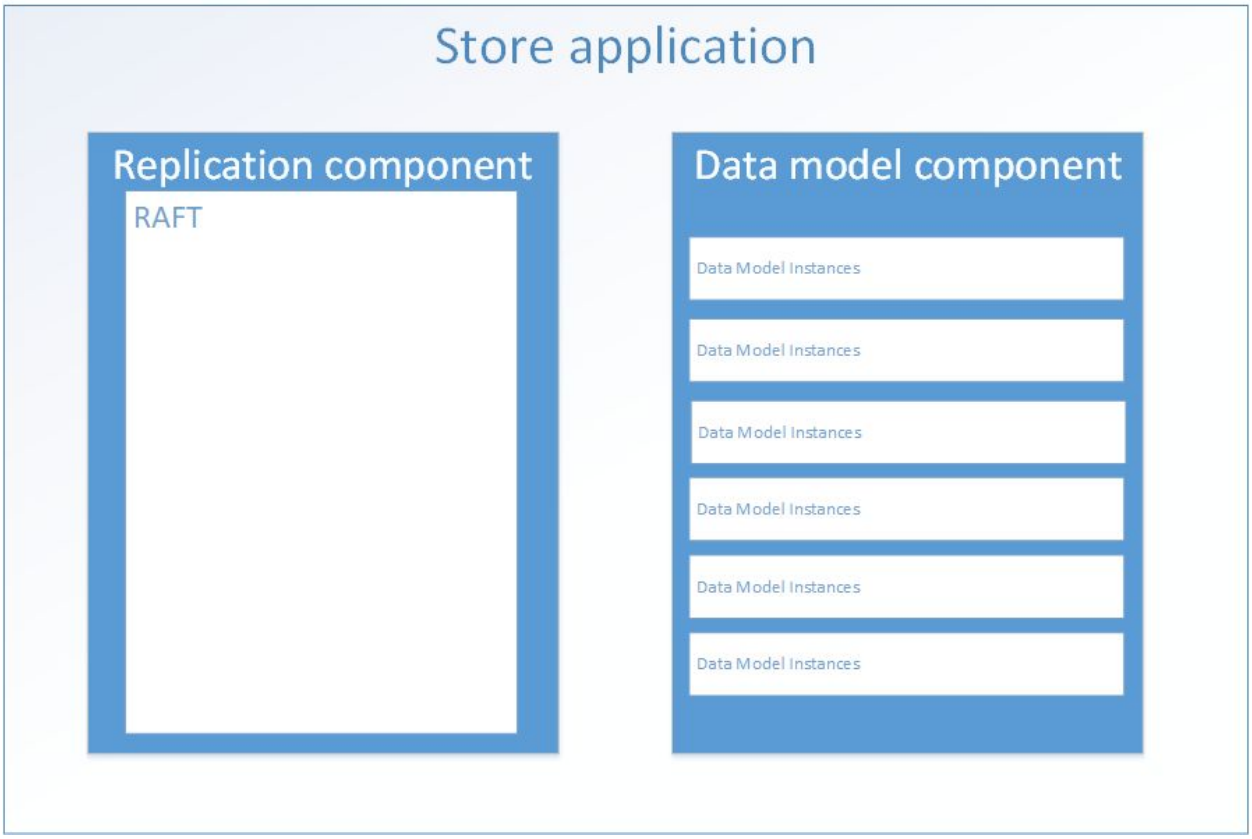
One of the main differences between the store application running on a master node[2.2.4.3.] and a slave node[2.2.4.3] within a single replication group is from whom they receive requests. When an application runs on the master node, it receives the request from the clients directly and raft specific messages from other nodes within its replication groups. However, when it runs on the slave nodes, it receives only the raft messages from other the replication group nodes. Another notable difference is that, the slave node application receives only modifying requests since the master node application processes it without replicating to the slaves. When we speak about a master and slave in the following sections, we always think of them in a context of a single replication group.

In the following subsections, we divide the functionality of the store application into two subcomponents, and then we describe their functionality, responsibilities, and interaction between them. Next, we focus on how the store application persists client's data and how it caches part of it data in its main memory. Finally, we describe how the application subcomponents interacts with our persistence mechanism.

5.1. Store internals

Conceptually, we can divide the store application into two components, a replication component and a data model component. The replication component implements the consensus algorithm RAFT[2.2.4.3]. The second component, data model component, manages the *data model instances*[1.1.2.1]

Following diagram depicts this division into components.



5.1.1. Replication component

As we already stated, the replication component implements the RAFT [2] algorithm, that means it is responsible for keeping the state of replicated data model instances within a single replication group in a valid state. This component has different behaviour when it runs on a master, and when it runs on a slave[2.2.4.3].

Replication component receives the request from network, and when it is sure that the client request is replicated within the replication group, it passes the client request to the data model component. We refer to replicated client requests as *committed* client request further on.

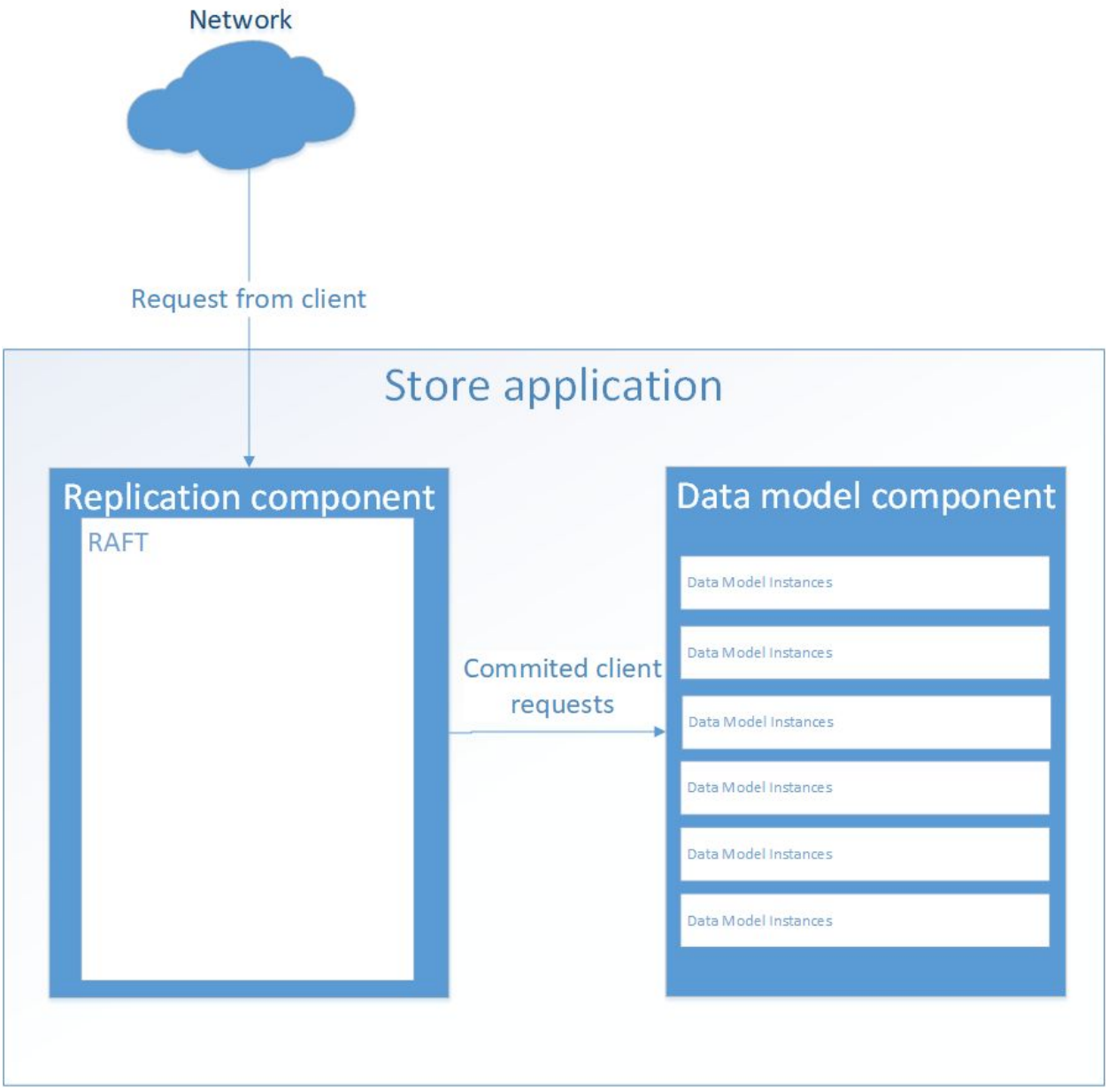
5.1.2. Data model component

The data model manages data model instances and passes them the client requests from the replication component.

For example, if we have an instance of key-value map inside the application and the replication component passed a request to insert a new mapping into to the map, then the data model component determines from the client request the data model instance, the request belongs to, which is the key-value map instance. Then the request is passed to the key-value map instance, and the key-value map instance inserts the new mapping.

The behaviour of this component is the same for both master and slave. This component is isolated from the replication itself.

Following diagram depicts the data flow of client request inside the store application. It shows how the *replication component* is processing data from network, be it from the other nodes in the same replication groups or the client. Furthermore, we can see that the *replication component* passes then the *committed* client requests to the *data model component*.



5.1.3. Threading model

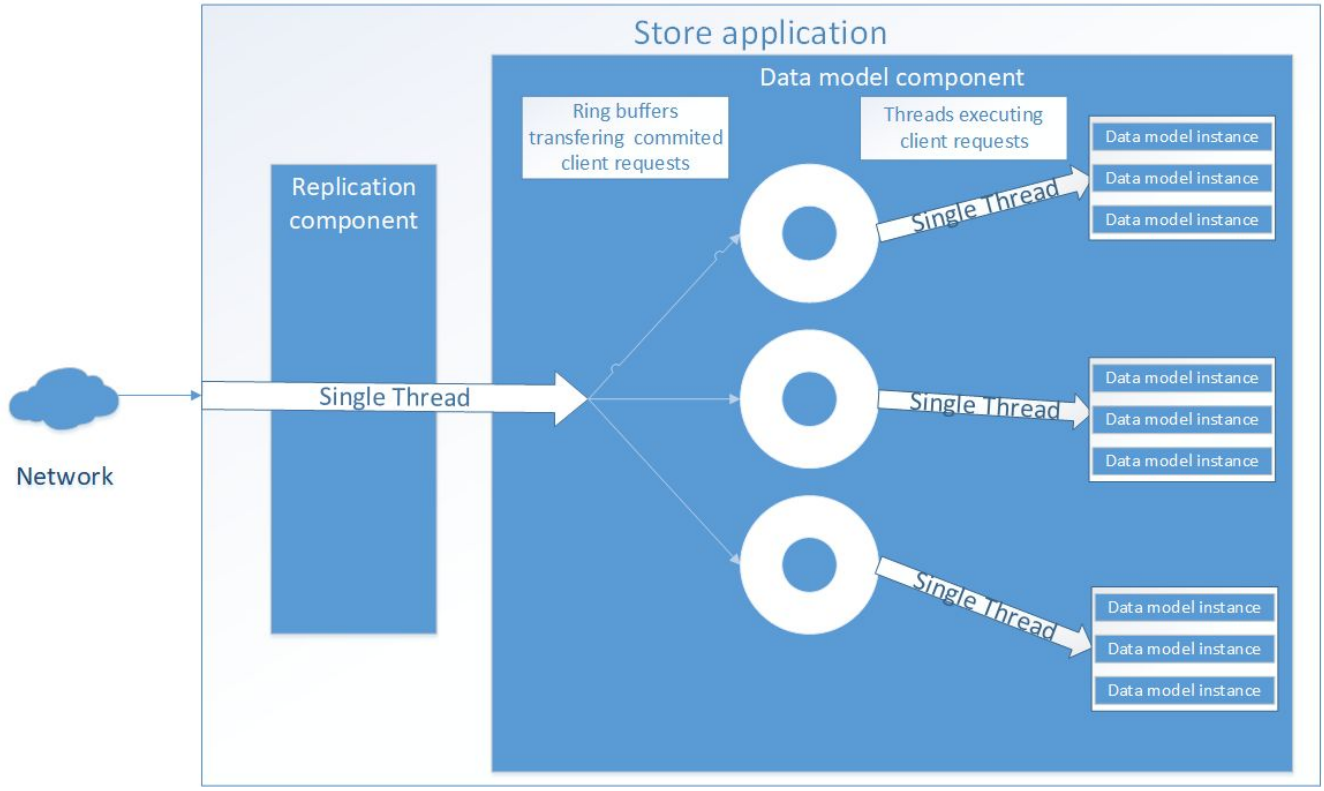
Since the performance of the store application is important, we want to utilize the multicore cpus and run multiple threads within our application.

The replication component runs in a single thread. In contrast, the data model component is multi threaded. So, the *replication component* within each store application receives messages from network, processes them, and passes the *committed* request within in a single thread. In the *data model component*, each thread processes client requests for multiple data model instances. Additionally, each *data model instance* is accessed only from one of the *data model component* thread.

When the *replication component* passes a client request to the *data model component*, it is done from the replication component thread. In the same thread, the data model component finds out on which of its thread the data model instance, the request belongs to, lives, and the request is then pass to that thread.

We implement the whole client request flow and the passing of requests between the application components using our *core.reactive* library [3.2]. So for example, the passing the data from the replication component thread to one of the data model component threads is implemented via the *FlowConnectors* [3.2.2.1], which are backed by ring buffers. We use the *core.reactive* library because it is convenient for the implementation.

Following diagram depicts the threading model of the store application. It also shows each thread responsibilities and their interaction with one another. Most notably, we can see the communication between the *replication component* thread and the *data model instances* threads which we implement with the ring buffers.



The chosen threading architecture has one distinct feature that is not used in a more traditional approaches multithreading. We minimize the sharing of data between threads. The two most obvious examples of this are how the replication component is always accessed from one thread only so is each data model instance. We took an inspiration from the Actor Model [11] when designing the threading model for the store application. The main difference between our design and the actor model, is that we have a small number of threads and during the application run this number rarely changes. In comparison, there can be many computation entities, so called actors, in actor model. What we borrow from the actor model, is the idea of minimizing sharing data.

As we can see, our threading model is relatively simple. We avoid sharing data between the threads as possible, use message passing for communication between threads and have relatively low number of them it the application, generally less then the number of cpus on a given hardware.

5.1.3.1. Sharing data among threads

We avoid sharing data among threads because it requires the synchronization of accesses from different threads. As a result, we implemented the *replication component* as a single threaded component and let each *data model instance* be accessed from a single *data model component* thread. Traditionally, locks are used to implement synchronization.

While using locks is sufficient in terms of performance for most application, in our case they negatively affects the performance of the whole system due to lock contention. Lock contention can block execution of multiple thread at once; thus, it has negative impact on a performance of the application. Another disadvantage is the implementation overhead of using locks or generally implementing synchronization. The usual recurring synchronization bugs are deadlocks and race conditions. In our experiences with implementing multithreaded applications, these bugs are hard to avoid, difficult to find, and slow to resolve.

In comparison, in our approach we don't share any data among threads. Therefore, we avoid race conditions, deadlocks, and lock contention . When one thread needs to access data from another thread within our application, it passes message to thread owning the data, and then it receives response with the data back.

5.1.3.2. Inter thread communication

The only place in the store application, where threads communicate with each other, is inside the data model component when the thread, processing networks request, passes committed client requests to the data model instance running another thread. We already stated that the passing of the committed client requests is implemented using the *core.reactive* library that internally uses ring buffers for communication between threads.

In the list third party libraries [3.4.1], we specified that we use two libraries, which already contains ring buffer implementations, Disruptor and Agrona. Both libraries contains a lock free ring buffers. One of these ringbuffers is used to in the *data model component* when transferring the *committed* client request.

Another advantage is that a *replication component* thread doesn't block when it sends the *committed* client request to data model instance thread in most cases, because the ring buffers are lock free. The sending thread blocks its execution only when the ring buffer is full and that happens only when one of the data model instance thread cannot keep up.

5.1.3.3. Alternative approaches

When designing the threading model of store application, we considered another alternative, such as running everything in a single thread or using thread pool.

If we choose to implement the whole application within a single thread, the performance would suffer. For example, a single client request, which is computationally expensive, can block the execution of everything else.

Another approach is using a thread pool, such as java *ExecutorService* [12]. This approach usually requires some form of a task based system, i.e. there must be a class which wraps abstract the execution inside the thread pool, such as java *Runnable* [13]. This class must be allocated. In managed languages such as java or c#, allocation eventually results in garbage collection. This pattern also does not deal with synchronization of accesses to the data.

5.2. Persistence and caching in main memory

5.2.1. Persistence

As we already stated at the beginning of this document[1.1.1], the store application persist all data to the disk. Additionally we discussed that raft uses log to ensure the consensus[2.2.4.3]; moreover, we stated that this log must be persisted on the disk to ensure the fault tolerance. Each entry inside this log contains serialized requests from the client. We will refer to this log as *raft log* further on.

Since the *raft log* contains all replicated client requests, it must also contain all the stored data within the store; therefore, the application persists all data saved within the store by persisting the *raft log*. We choose this approach because we do not see any clear benefit to persisting the data outside of the raft log. Such a solution wastes the disk space, and it adds an implementation overhead.

This diagram up shows the example of a *raft log* persisted on the server disk. We can see that the raft log contains three serialized client requests for some *data model instances*.

5.2.2. Raft log in memory

Raft log is persisted on a disk but a part of it is also cached in the memory. We keep the portion of raft log in the memory for various reasons.

- Caching a portion the raft log in the memory yields better performance, because we do not have to do the IO every time we want to access an entry inside the log.
- The entries in the raft log are used to pass the committed client request of the data model instances. We are going to describe this in more details.

In the next subsection, we describe how the data model instances interacts with the *raft log* and the rationale behind the described design.

This diagram depicts the whole *raft log* persisted on a disk but in contrast to the diagram in previous subsection, portion of the *raft log* is also cached within the main memory.

5.2.2.1. Raft log and Data model instances

Since we append the client request to the *raft log* as a part of the RAFT algorithm, the replication component is responsible for it. When the replication component passes the client request to the data model component, which then passes it to the data model instance, the replication component does not pass the request as an instance of some java class or serialized into a byte array. Instead, it passes to a memory pointer to the request inside the *raft log*. To make sure the passed pointer is valid, we guarantee, that when an entry, containing the client request, is appended to the log, it is cached in the main memory.

We pass the pointer to the log entry because in java we represent a pointer as long, which is a primitive type. So passing the pointer does not generate garbage as opposed to passing an instance of some java class. In addition, passing the client request as an instance of java class would complicate the design of data model interfaces. If we only pass the memory pointer to the data model, its implementation can reinterpret the serialized client request in any way it likes; moreover, this reinterpretation is an implementation detail. This is more flexible approach then passing the client request as an instance of some class.

Another approach, we could choose is to serialize the client request into a byte array and then pass that byte array to data model instance. While this approach is feasible, it involves unnecessary serialization and deserialization of the client request. Additionally, the raft log already contains the serialized client request. There is no clear benefit of passing the client request serialized into the byte array.

Finally, the data model implementations don't need to store all information from each client request. When they receive the pointer to the log entry containing the client request, they can extract all relevant information from the request and store it inside their internal structures. All other informations, can be later accessed via the pointer to the log entry.

5.2.2.2. Caching raft log in memory

As the application runs, the *raft log* will grow in size. Hence, it is impossible to cache the whole log inside the main memory. We have to provide a mechanism, which keeps only portion of the *raft log* cached in the main memory.

We delegate the responsibility for choosing which log entry stays in the main memory and which is only persisted on a disk to the data model implementations. Since the data model instances works directly with the log entries, they can easily control which log entries are cached and which are not. We provide the data model implementers with the API to control the caching.

This approach has a good performance given that the data models are reasonably implemented and it is not hard to implement. The disadvantage is that it gives a lot of power to the data model implementations.

One of the alternative approaches would be keeping only the last few log entries cached in the main memory. However, this approach does not yield in good performance since the frequently accessed part of the *raft log* does not have to be at the logs end. Another approach would determine which part of the *raft log* are frequently accessed and kept those parts in cached. This approach have a good performance; however, it is hard to implement.

5.2.2.2.1. Default data models caching mechanism

As we stated previously [4.3], we provide two data models out of box, a list and a key-value map. Both of these data models have a mechanism for caching *raft log* in memory.

The list data model implements simple mechanism; it always keeps in the memory the *raft log* entries of the last *n* of its elements where n is configurable integer that is greater or equal to 0.

The key-value map data model contains a more sophisticated mechanism. It keeps in memory the *raft log* entries, which corresponds to the currently stored mapping within the map. For example, if we have a key-value map which contains the mapping with key *1* and value *bar* and we receive a committed client request to insert a new mapping with key *1* and value *foo*, the plugin then release the *raft log* entry with the mapping (1, bar) from memory and it keeps the new *raft log* entry with the new mapping (1, foo) in memory.

5.3. Summary

In this section, we have described how the store application works. Firstly, we divided the store application into two components; we describe their functionality and responsibilities. Secondly, we discussed how the threading model of the application. And finally, we describe how the persistence works.

6. Client library

Client library is used for a communication between client and the cluster. This library allows client to send read and write requests to the store.

When the library is initialized, first it reads properties file that contains configuration of the store cluster and tries to connect to all replication groups inside the cluster. The library sends request to each master using bidirectional multicast channel of type 3[2.3.1.]. Then it receives answers from masters containing all necessary information that is needed to establish all communication channels of type 1. Those channels are bidirectional unicast that are used by client library to send queries to the store and receive responses from the masters.

If new master is elected in the group, it sends message to all clients using type 3 connection, at this point client library closes old communication channel of type 1 to the old masters and opens new channel to the newly elected master.

6.1. Data model communicators

A data model communicator allows the client to work with a data model instance stored in the cluster. The API and functionality this communicator is specific to particular data model. For example we are using 2 types of data model communicators per each default data model[4.3.]: list communicator and key-value communicator. Client can use those communicators to read, write, update or remove items from the store.

These data model communicators provide the client functions to work with concrete data model instances inside the store. Underneath communicator transforms, each function call to the binary request and the client library sends the serialized request to cluster. In addition for each request, except for the queries, data model communicators passes to the client library which replication group [2.2.1] receives the serialized request. The client library sends the serialized binary data with its own header. This message is then send by the network layer to the replication group master using communication channel of type 1.

The client library provides an easy mechanism to integrate with new data model communicators. It is the same mechanism used to integrate new data model plugins[4.2]. The client library, also, has an API to create new data model instances and to remove the existing ones.

Aeron[3.3] will be used as mean of communication between client library and cluster. Client library is use *AeronSender* to send client requests to masters, and *AeronReceiver* is used to receive response from the masters. Aeron messages are created in the client library using binary data from data model communicator.

6.3. Threading Model

There are two threading models for the client library to communicate with the cluster. One is used for situation where the performance is not needed, and the other is used for high performance scenario. Client applications choose the threading model of the client library before they start communicating with the store itself.

6.3.1. Low performance mode

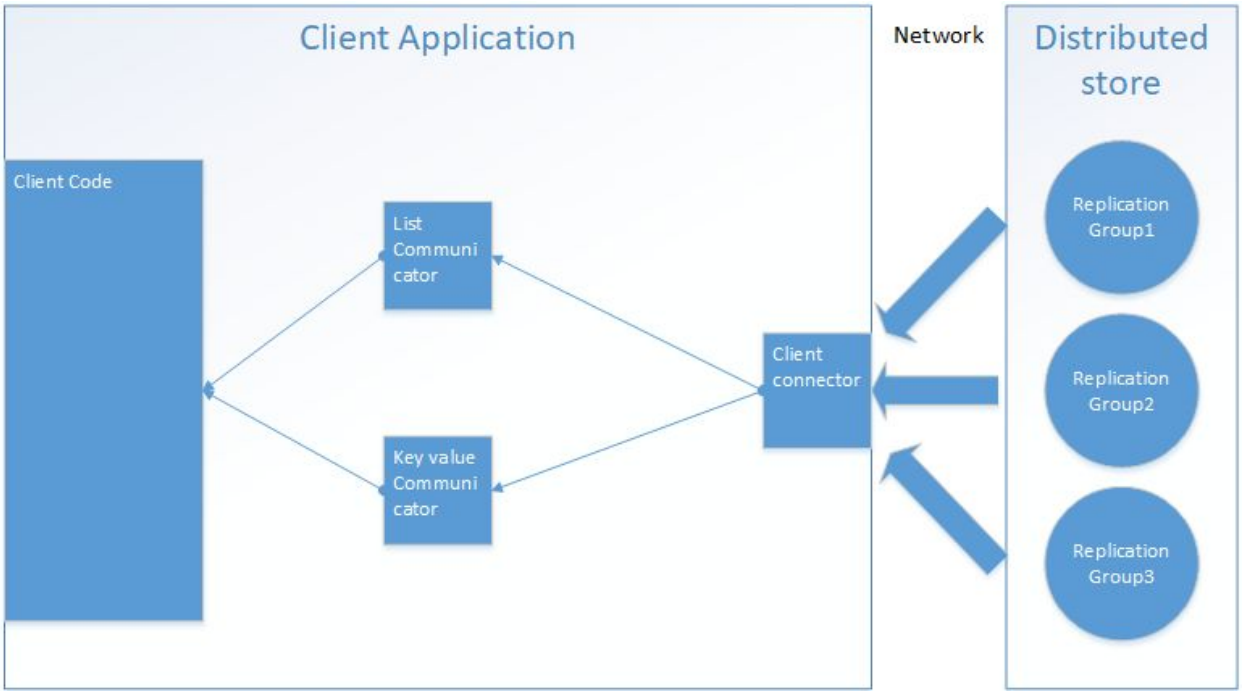
6.3.1.1. Sending data in low pefomance mode

Client library uses two threads. One for sending queries to each cluster group inside the distributed store and one for receiving messages from masters. Data model communicator sends all binary requests and its destination to one instance of client connector. Then this instance is responsible to send those requests to each master in the distributed store using Aeron.

This threading model is used when it is needed to save the hardware resources and low performance scenarios. In addition, we use this model for debugging and testing purposes. This model is shown in the diagrams below.

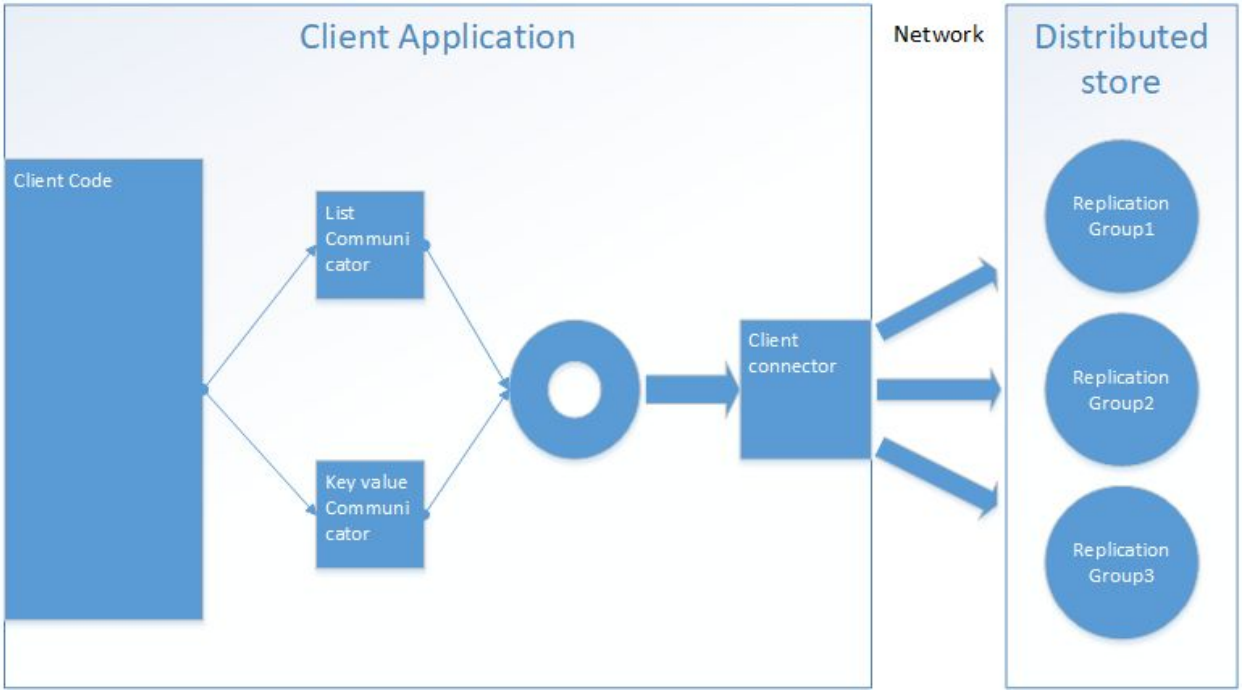
6.3.1.2. Receiving data in low performance mode

Data model communicator receives responses in the same thread as client connector is receiving responses from the replication groups. We decided to implement the receiving this way because the communicators can choose whether they process the response on client connector receiving thread, or whether they process the response in a separate thread. If the communicators decides to processes the responses in another thread, they must transfer the response themselves from the client connector receiving thread. Each time connector receives the response from the replication group that response is sent to the data model communicator. There it is decoded and is passed to the Client application. Following diagram shows the flow of data when receiving responses from distributed store.



6.3.1.3. Implementation

Since client connector is using one thread for sending request single thread for sending requests, communicator is transfering requests to that sending thread through a ring buffer. The implementation will be defined using core reactive library [3.2].

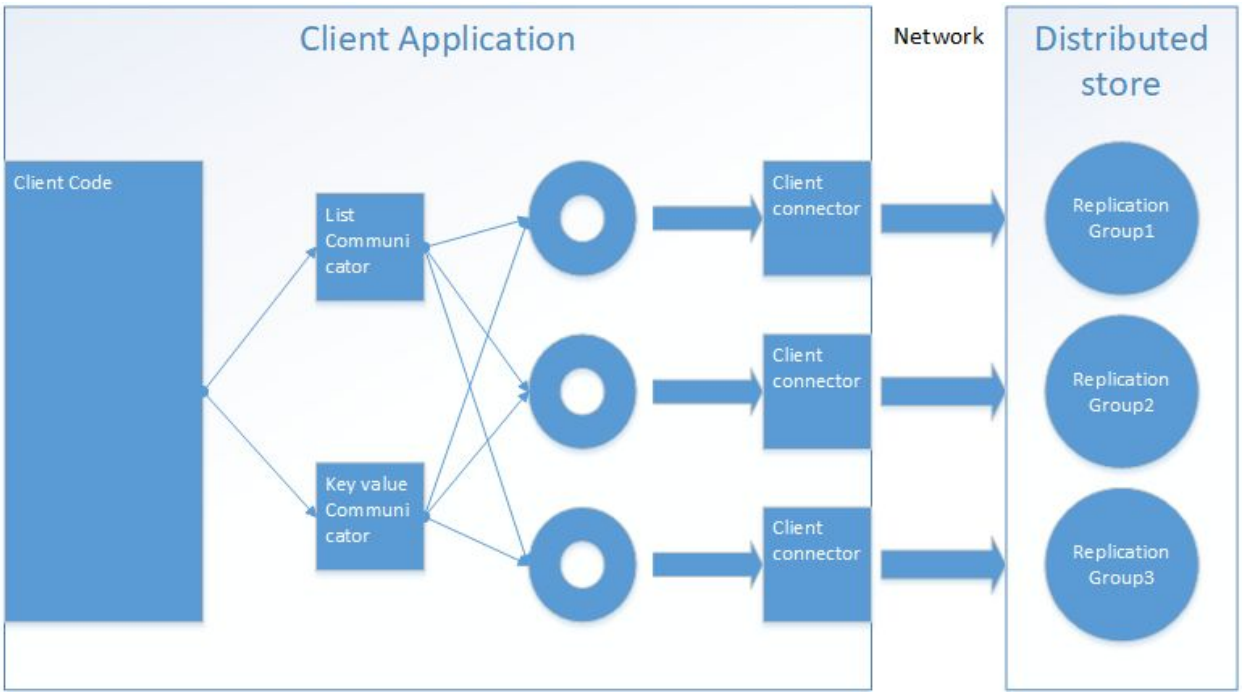


6.3.2. High performance mode

6.3.2.1. Sending data in high performance mode

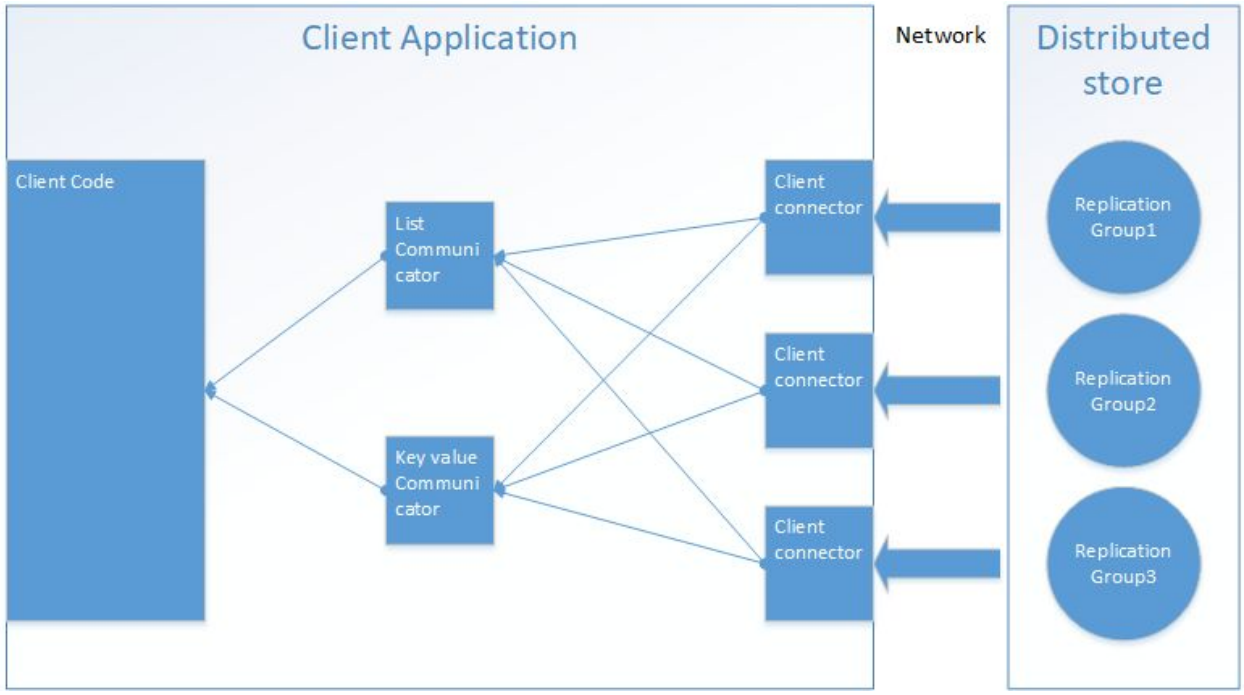
Client uses two threads for each replication group inside the store. One is used for sending requests and one for receiving responses. The same way as in low performance mode, for each request from the client data model communicator knows to which replication group it should go. It is transferring request through the corresponding ring buffer to the client connector that is connected to the correct replication group. In contrast to low performance mode, the client connector instance is not responsible for choosing to which replication group it is sending requests, it is connected to a single replication group and is constant for the whole run.

This use case is harder to test. It is used for high performance scenarios. This threading model is closer to real life use case. This model is shown in the diagram below.



6.3.2.2. Receiving data in high performance mode

Receiving data in high performance mode is similar to the low performance mode, with the only difference being that each replication group is sending response to the corresponding client connector. The data flow is shown in the diagram below:



6.3.3. Non-blocking API

We use the same principle just like in the case for the Store application where we minimize the sharing of data among threads. We have threads that are responsible for the communication with the store and other the client application threads communicates with the communication threads via the ringbuffer. Consequently, the client library has a non-blocking api.

7. Functionality integration tests

7.1. Demo client application

Demo client application is a simple console program that will be used for testing the whole system. It will be using client library to communicate with the distributed store. This application will simulate user manipulations with the store during integration and performance testing.

The client application runs multiple testing scenarios and has an easy mechanism for integrating new testing scenarios. We provide at least few scenarios as a part of the project; those scenarios are used for our internal testing.

7.2. Functional and integration tests.

In our tests, we simulate usage, which is similar to a real world use case. We store messages that are used for a communication in a financial industry between different parties. For example, messages exchanged between the customers and the exchange during trading. These messages usually contain a key-value structure, where the key is an integer and value have multiple types, such as strings, integers, float etc. For our tests, the message also contains key-value structure, but value are only integer. This is simplified use case but is enough as a proof of concept.

Integration test were set up on the early stage of the development. Every day during night they are ran to confirm system's integrity and stability. Test scenario is 3 clusters each contains 3 nodes and client application that connects to them and sends read/write queries.

7.3. Test scenarios

Several use cases will be tested in the integration tests:

1. Tests of the consistency of the store. First fill the distributed store with data and then check if the data inside is correct.
2. Failing master during sending of the data, and testing the consistency of data. Test that simulates changes inside replication group. For example if the master is changed. + If store can cope with the loss of one of the cluster nodes.
3. Additional tests for querying the distributed store. We fill store with data and then start sending queries, checking if the result is correct.

8. Benchmarks and pcap analyzer

We use pcap analyzer for evaluating the performance of the store. The pcap analyzer provides us with information about the latency and throughput from the benchmark of the store. In addition to the pcap analyzer, we also discuss how we benchmark the store in this section.

8.1. Benchmarks

We run two set of benchmarks. The first set of benchmark focuses on testing the throughput of the store, and the second set focuses on the latency of the store. We execute all our benchmarks in *Rapid Addition* performance lab.

8.1.1. Throughput benchmarks

This set of benchmark is designed to test the throughput of the store, i.e. how many client request is the store able to handle per specific time period, such as 1 second.

In each of these benchmarks, we deploy the store and few instances of the client demo application into the performance lab. These instances then simulates the clients. During the benchmark, the simulated clients are issuing request to the store and we analyze how many of these request the store is able to handle per 1 second.

8.1.2. Latency benchmarks

In the latency benchmarks, we measure a latency of a client request, i.e. the time between sending the request to the store and receiving the confirmation about the request execution from the store on a client side.

As for the throughput benchmark, we again deploy the store and few demo client applications, which simulates the clients. Then, we measure the latency of client request on each of the simulated clients.

8.1.3. Benchmark environment

The *Rapid Addition* performance lab contains few high spec servers which are connected by a high quality network; moreover, this lab has a specialized network tap which captures all the traffic on the network without any packet lost. An example of such a device is solarflare solarcapture [14]. We can use this network capture to sniff the traffic packets into a pcap file and this pcap file can be later analysed.

8.1.4. Benchmark methodology

When we run the benchmark, we do not have a special software component which analysis benchmark output and processes them. Instead, we capture all the traffic on the network during the benchmark using the network tap into a pcap file, and we analyze the captured pcap to get the benchmark results. We use pcap analyzer to extract all the relevant informations from the pcap.

8.2. Pcap analyzer

Once we execute any of our benchmarks, and we captured the network traffic into a pcap file from that benchmark. We analyze the pcap file with the pcap analyzer. When the pcap analyzer finishes its execution, it outputs information about a throughput of the store or a latency for each client request during the benchmark run.

The pcap analyzer parses the captured pcap file. It extract the network packets from the pcap and then detects the network packets which belongs to the request sent from the simulated clients. Once, the packets are extracted, the pcap analyzer outputs information about the latency or throughput based on the hardware timestamps.

For measuring the latency, the pcap analyzer pairs the each packets belonging a client request with the packet belonging to the confirmation about execution of that client request. It computes the latency of this request as difference between the hardware timestamps of the request packet and it's confirmation packet.

When we measure a throughput, the pcap analyzer computes the number of confirmations for each client request per specific period of time for each simulated client. We, again, use the hardware timestamps for the computation.

8.2.1. Extracting the client request packets

Detecting the packet belonging to client request from the pcap [5] is the basic functionality of the pcap analyzer. As we already discussed, our whole communication is build on Aeron middleware [3]. The client request and the confirmations are transferred via aeron packets. Therefore, to measure the latency, we must extract these packets from the pcap file.

Pcap file contains list of pcap packets. Each of these pcap contains an header and a packet data. The header contains the hardware timestamp and the packet data contains in our case the ethernet packets. From which, we must assemble the ip packets, and then we assemble the udp packets from the ip packets since Aeron works on top of udp. Once we have the udp packets assembled, we can build the aeron packets from them. Finally, the built aeron packets already contains the client requests.

Implementing the packet assembling is not trivial; therefore, we use the Pcap4j library, mentioned in section 3.4.1. This library help us with assembling the udp packets from the pcap records. Once we have the udp packets, we use Aeron internal api to assemble the client requests from aeron packets.

9. Harmonogram

We expect that we finish the whole project within the 7 months in different phases, and we already spent two months preparing the specification. This document contains the harmonogram for the remaining 5 months.

We splitted the harmonogram into 4 different phases. Each phase has its goals and requirements. In addition, we specify tasks for each member within each phase. By finishing each phase, we reach an important milestone in the project that helps us in our progress.

Next, we list the phases in chronological order; we specify for each phase goal and functionality requirements, additionally, we try to establish estimates for the tasks in man-days for each team member.

9.1. Development infrastructure

In the first phase, we set up our development environment for which includes:

- Setting up continuous integration (CI) system.
- Setting up git repository.
- Ticket tracking system.
- Configuring IDE.
- Creating the java project.
- Setting up build project for the java project.

We expect finish this phase within 2-3 weeks. We choose jenkins as our CI system because we are most familiar with it. We deploy an instance of jenkins to the cloud. Additionally, we use the bitbucket as our git repository.

9.2. Implementation of core library and prototypes

There are two main goals in this phase. First, we implement a large portion of the core library, and then we create a prototype of all three applications, which can already build on top of the core library. We estimate that this phase will be finished within 1.5 month.

9.2.1. Core library tasks

We can split the task for implementation of the core library into multiple categories, the di framework based on guice, integrating the Aeron middleware, implementing the core.reactive library, and implementation of various utilities such as config parsing. We split task among each team member.

9.2.1.1. DI framework based on guice

The DI framework is described in the section [3.1]. There are two task to finish this, implementation of the wrapper upon *Guice* and loading of application plugins[3.1.2.2].

Task	Implementer	Estimate
Wrapper around Guice	Ladislav Maleček	2 MD
Loading of application plugins	Tran Tuan Hiep	4 MD

9.2.1.2. Integration of Aeron

The core library contains utilities communicate via Aeron middleware[3][3.3]. It contains *AeronSender* for sending messages via Aeron and *AeronReceiver* to receive messages from network.

Task	Implementer	Estimate
Aeron Sender	Anton Khodos	2 MD
Aeron Receiver	Ladislav Maleček	2 MD

9.2.1.3. Core.reactive library

Implementation of *core.reactive* library is splitted is also splitted into two tasks[3.2]. We need to implement the *FlowConnectors* and operators. *FlowConnectors* are responsible for communication between threads and operators are used transform events[3.2.2.].

Task	Implementer	Estimate
Flow connectors	Tran Tuan Hiep	5 MD
Operators	Anton Khodos	2 MD
Operators	Andrea Turčanová	2 MD

9.2.1.4. Other utilities

This last part of the core library contains the usual utility such as configuration parsing, various utilities for multithreading, debugging, logging, and so on.

Implementer	Estimate
Andrea Turčanová	3 MD
Ladislav Maleček	3 MD

9.2.2. Prototype tasks

In the first phase, we also want to create prototypes for the applications, the store application [5], demo client application [7.1], and pcap analyzer [8.2]. We mainly focus on the first as two since they are more important for integration tests. Each prototype relies on the functionality implemented inside the *core library*.

9.2.2.1. Store application prototype

The store application prototype is simplified version of the application [5]. Unlikely from the actual application, the prototype does not implement consensus algorithm, and it does not store any actual data. However, it already uses *Aeron* middleware [3] [3.3] and supports the dependency injection framework Guice [4][3.1].

When the prototype runs in a cluster of nodes, there is always one master node and all other nodes are slaves. I.e. there is only one replication group[2.2.1]. The master node is responsible for receiving requests from client prototype. The master node processes each request from a client application prototype, and it sends this request to the slaves, which confirms the request. Once the master receive at least half confirmation from half of slaves. It sends a confirmation to the client application prototype.

When we deploy this prototype, we preconfigure which node is a master and which is a slave.

Implementer	Estimate
Tran Tuan Hiep	3 MD
Anton Khodos	2 MD

9.2.2.2. Demo client application prototype

The prototype of this application is a very simple. When it starts, it connects to the master of the *store prototype*. Then, it sends requests to the master and receives responses from it. In addition, it relies on *Aeron* and *DI* framework like the store prototype.

Implementer	Estimate
Andrea Turčanová	3 MD

9.2.2.3. Pcap analyzer application prototype

This prototype contains basic functionality, such as parsing the pcap records from the sniffed pcap file. As previously, it supports the DI framework.

Implementer	Estimate
Ladislav Maleček	2 MD

We expect the first phase to take us approximately month and half.

9.3. Finalizing the implementation

In this phase, we finalize the implementation of our application. We extend the prototypes created in previous phase so that they conform to their specification. Additionally, we also set up our integration test within this phase.

9.3.1. The store application

We need to implement the consensus algorithm *Raft* [2.2.4.3], the default data models[4.3], and the persistence mechanism[5.2].

9.3.1.1. Implementation of Raft

The implementation of Raft is probably the most difficult from all subtasks; we expect to finish it within 7 MD. We split the task between two team members.

Implementer	Estimate
Tran Tuan Hiep	10 MD

Anton Khodos	3 MD

9.3.1.2. Default data models

We need to implement two data models: key-value and list data models[4.3]. The implementation itself should not be hard.

Task	Implementer	Estimate
Key-value data model	Anton Khodos	2 MD
List data model	Andrea Turčanová	2 MD

9.3.1.3. Persistence mechanism

Our persistence mechanism is closely connected with the Raft implementation[5.2.1], because it relies on the *raft log*. This task consist of integration of *raft log* within the data models and the implementation of memory [5.2.2.2].

Task	Implementer	Estimate
Persistence	Tran Tuan Hiep	4 MD
Caching *raft log* in memory.	Anton Khodos	5 MD

9.3.2. The client library and the demo client application

We need to implement the *two threading model*s and *data mode communicators* to finish the client library. Additionally, we also extend the prototype of *demo client application* from previous phase.

Since the prototype contains functionality of both the application itself and the client library, we start with splitting the prototype into two parts.

9.3.2.1. Client library

We move a network communication based on Aeron to the client library. Then, we implement the inside the threading modes [6.3] and integration of *data model communicators* for default data models [4.3].

Task	Implementer	Estimate
Aeron communication	Andrea Turčanová	3 MD

Threading modes	Andrea Turčanová	4 MD
Threading modes	Ladislav Maleček	3 MD
Key-value data model communicator	Anton Khodos	2 MD
List data model communicator	Andrea Turčanová	2 MD

9.3.2.2. Demo client application

The demo client application is used for integration tests and as an example of integrating and working with the client library. [7.1]. Moreover, this application supports plugging multiple testing scenarios, which are used for integration tests and performance tests.

Task	Implementer	Estimate
Integration of client library	Andrea Turčanová	2 MD
Execution of testing scenarios	Anton Khodos	4 MD
Implementation of the testing scenarios	Ladislav Maleček	4 MD

9.3.3. Pcap analyzer

Implementation of the pcap analyzer is straightforward unlikely from the previous two implementation. We only need to extend the analyzer prototype so it conforms to the specification. [8.2].

Task	Implementer	Estimate
Parsing of pcap records into Aeron packets	Ladislav Maleček	7 MD
Implementing evaluation modes	Tran Tuan Hiep	2 MD

9.3.4. Integration tests

At the end of this phase, we already have functioning applications; therefore, we need to set up integration tests. These integration tests will run each night.

Implementer	Estimate
Anton Khodos	2 MD

9.4. Fine tuning and documentation phase

In the last phase, we mainly focus on bug fixing, finishing documentation, and benchmarking. We expect this phase to take us 1.5 month.

We estimate to spend 20 MD together on bug fixing. Additionally, we extend this specification into a final documentation; this should take at least 10 MD. In addition, we also set up the benchmarks and evaluate the performance of the store. More information about the benchmarks are in chapter 8.

Task	Implementer	Estimate
Bug fixing	Each team member	20 MD together
Documentation	Each team member	10 MD together
Benchmarks	Anton Khodos	1 MD
Evaluation of benchmarks	Tran Tuan Hiep	2 MD

10. Summary

We described each aspects of our software project in this document. At first, we gave a general overview of our project. Then, we described the design of distributed store cluster in the second section. That section focused on the architecture from a higher-level point of view. In following sections, we analyzed each of our existing software components, the core library, store application, client library, demo application, and the pcap analyzer. For each of these components, we gave a detailed description of their functionality, explained their design and provided the rationale behind their design.

References

[1] L. Lamport, "The part-time parliament," 1998.

[2] O. D. a. O. John, "In Search of an Understandable Consensus Algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, 2014.

[3] Aeron. [Online]. Available: https://github.com/real-logic/aeron.

[4] Google, "Google Guice," [Online]. Available: https://github.com/real-logic/aeron.

[5] "Libpcap format," [Online]. Available: https://wiki.wireshark.org/Development/LibpcapFileFormat.

[6] Oracle, "ServiceLoader," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html.

[7] "ReactiveX," [Online]. Available: http://reactivex.io/.

[8] ReactiveX, "RxJava," [Online]. Available: https://github.com/ReactiveX/RxJava.

[9] R. Streams, "Reactive Streams," [Online]. Available: http://www.reactive-streams.org/.

[10] R. manifesto, "Backpressure," [Online]. Available: https://www.reactivemanifesto.org/glossary#Back-Pressure.

[11] "Actor model," [Online]. Available: https://en.wikipedia.org/wiki/Actor_model.

[12] Oracle, "ExecutorService," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html.

[13] Oracle, "Runnable," [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/lang/Runnable.html.

[14] Solarflare, "Solarcapture," Solarflare, [Online]. Available: https://www.solarflare.com/solarcapture.