Distributed Store

OVERVIEW

In this document, we describe software project which we intend to propose for subject NPRG023 at Charles University in Prague. We, the team of 4 students of the university, will complete this project. Our team will implement a distributed data store and library to communicate with the data store, 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.

The project will serve as a prototype for Rapid Addition Ltd. for the their own distributed data store.

GOALS

The project consist of four application components, that we must implement.

1. Distributed data store
    - The main component of the whole project.
    - Multiple clients work with the store. They send the request to read or write data request to 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, that will serve as an example of the integration of the client library.
    - We will mainly use it for 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.

We consider the project successful, when we will have implemented all four application according to their specification. We will additionally provide an performance analysis of throughput and latency of the distributed data store.

SPECIFICATION

We will implement the whole project in Java. Rapid Addition requires an implementation in Java and using gradle as a build system. In addition to the existing four application, we will also create a common library which will contain a functionality used in all four components, such as networking, configuration parsing, and various utility classes.

Distributed store

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.

The store provides the clients with two data models, a key value store and a list, and clients store their data in these two data structures. We implement this data models inside the store as plugins; furthermore, our store have a plugin based mechanism for an easy integration of a new data model plugins. When running, the store has multiple instances of data model plugins; these instances stores their data inside data structures. For example, we can have a distributed store, with 3 instances of key-value map plugin and 2 instances of list plugins and all instances stores the data in isolation from each other.

The store supports querying on the stored data but we don't implement the querying globally in the store; rather, we delegate the querying implementation to the data model plugins. We choose this plugin based approach for querying opposed to having a single mechanism within the store because the plugin can implement querying that is both specific and highly optimized for their stored data.

The store eventually persists all its data on the disk; however, the data model plugins can control which data are written on disk and which are cached in the main memory. The distributed store have a reasonable performance, when the cached data in memory on every node fit in it's main memory.

The store ensures a replication of a each store data item on multiple nodes.

Data replication

We split the store nodes into multiple groups to support the replication. Each of these groups stores different set of data. Each node replicates the same data in a same group. Nodes from different groups cannot store the same data item.

The data store can cope with a crash of a node within a single group. Data, stored on the crashed node, are replicated on the other group nodes.

We implement a RAFT consensus algorithm to ensure a consistency among individual replicas in a single group. RAFT guarantees a strong consistency; however, we may modify the algorithm and weaken the consistency. Our distributed store has its own implementation because we have not found an implementation of raft that would meet our performance requirements, such as being GC free.

RAFT

RAFT ensures a consistency for a distributed state machine.

We can imagine each node of a group as a state machine. This state machine manages data stored in that node. Data content is a state of the machine and clients write request are transition from one state of the machine to another. RAFT ensures consistency among the state machines in the group by replicating a log of requests from clients in each node.

RAFT and the replicated log are a central parts of the store.

Implementation and technology specifics

The store must be GC-free, this is a core requirement. We heavily rely on sun.misc.unsafe to reduce the GC. As a consequence, the store implementation does not conform to the java language specification.

We implement most of the network communication with Aeron, low latency and high throughput middleware. We have a working experience with Aeron.

Our implementation relies on dependency injection framework Guice. Dependency injection enables high modularity and extensibility of the application.

We rely on a number of third party libraries such as:

- Agrona - collection of java high performance data structures
- Guava - extension of standard java libraries
- Log4j - logging

Our common library depends on these third party libraries. As with Aeron, we already have experiences with both the dependency injection framework Guice and the specified list of the 3rd party libraries.

Client library

Java library which implements communication with the data store. The library have the following functionality:

1. Parsing the config file with the configuration of the distributed store.
2. Establishing connection with all groups of the distributed store via the aeron middleware.
3. Notifying the client once it is connected with the store, e.g. connected with the groups.
4. Provides API for sending read and write request to the store.
5. Dynamically create new instances of data model plugins in the store.
6. Support for querying on data.

The implementation must be GC-free.

Client application

Command line application which tests the store functionality and showcases 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 have 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. Additionally, the application relies on dependency injection framework just as the distributed store.

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.

Pcap analyzer assembles the ip packets into the aeron packets, then assigns the packets to a specific request from clients.

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.

TASK FOR EACH TEAM MEMBER

- Tran Tuan Hiep
  - Core of the distributed data store implemented as framework
  - Common library
  - Coordination of other team members
- Anton Khodos
  - Core of the distributed data store implemented as framework
  - Common library
- Andrea Turčanová
  - Client library and demo client application.
  - Implementation of data model plugins working with the framework implemented by Tran and Khodos
- Ladislav Maleček
  - Pcap analyzer
  - Common library

TIMELINE

We expect that we finish the whole project within the 7 months in different phases. All phases have their goals and requirements and the tasks for each member are split into these phases. By finishing each phase, we reach an important milestone in the project that helps us in our progress. In our roadmap, we have four different phases.

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

Roadmap

Architecture and development infrastructure

- In the first phase, we have two goals, come up with the overall architecture of the project and to set up our development environment.
- The first phase take a month.

- We should have a document describing the overall architecture by the end of this phase. Coming up with an architecture and writing the document should take at least 4 MD for each.
- We setup our development environment, e.g. setting up the IDE, CI system, git repository, wiki. We may also set up ticket system such as JIRA.
- The estimate for setting up the environment

Tran Tuan Hiep	3 MD setting up environment
Anton Khodos	4 MD setting up environment

- At the end of this phase, we expect to be ready to start with the implementation.

Skeleton implementation and integration tests

- There are 3 main goals in this phase. We implement a common library and create a prototype of all 3 applications, i.e. the store, the client testing application, and the pcap analyzer.
- In the common library, we implement a module based application skeleton build upon the guice, aeron communication, config file parsing and various other utilities.
- The distributed store prototype does not implement RAFT and persist data instead it only imitates the raft consensus algorithm. We build the prototype on top of the guice module based application skeleton.
- The client application prototype only sends requests to the store and waits for the store response. In this phase, we try to implement basic functionality of the client application.
- The pcap analyzer prototype should already be able to extract aeron packets from pcap file; however, it does not output any meaningful data.
- At the end of this phase, we integrate all 3 prototypes in an integration test. The clients sends a multiple requests to the store which the store processes and the pcap analyzer only checks whether all client requests were propagated in the cluster. Ideally, we would be running this integration test each night.
- Estimates for each team member

Tran Tuan Hiep	5 MD common library + 3 MD distributed store prototype
Anton Khodos	3 MD common library + 2 MD distributed store prototype + 2 MD setting up the integration tests
Andrea Turčanová	3 MD client application prototype + 2 MD prototype plugin
Ladislav Maleček	3 MD common library + 2 MD pcap analyzer prototype

- This phase should take us at least a 1.5 month to 2 months.

Implementation

- The main goal of this phase is to finish the implementation of all 3 prototypes so that they work according to this specification.
- In the distributed store, we implement the raft protocol then we fully integrate it into the store. We create a plugin based framework in the distributed store, which is functional and ready for data model plugin integration, on top of the guice module application skeleton.
- We refactor the client application prototype into two components, the client library and an application that is build on top of the guice application skeleton from the common library.
- We create two data models plugin, list and key value store, and we integrate them into the distributed store framework.
- The data model plugins and the clients have a support for querying.
- After this phase, the pcap analyzer takes the sniffed pcap file and the distributed store configuration files. It is able to produce a information about a throughput and latency for each of the groups in the cluster.
- We start writing a documentation in this phase.
- At the end of this phase, we add new integration tests.
- Estimates for each team member

Tran Tuan Hiep	15 MD distributed store
Anton Khodos	3 MD common library + 5 MD distributed store + 4 MD integration tests
Andrea Turčanová	4 MD client application refactoring + 1 MD finishing the client application + 3 MD data model plugin integration into the distributed store
Ladislav Maleček	5 MD common library + 7 MD pcap analyzer

- This phase should take us at least 2.5 month to 3 months.

Fine tuning and performance evaluation

- This is a final phase, all 3 components should be ready after this phase.

- In this last phase, we mainly focus on bug fixing, performance optimizations, and finishing the documentation.
- Additionally, we evaluate the store performance.
- Estimates for each team member:

Tran Tuan Hiep	5 MD fine tuning + 1 MD performance evaluation
Anton Khodos	5 MD fine tuning
Andrea Turčanová	5 MD fine tuning
Ladislav Maleček	5 MD fine tuning + 2 MD performance evaluation

- We should have at least 1.5 month or 2 months for the final phase.

PROJECT SCOPE

Discrete models and algorithms	
	Discrete mathematics and algorithms
	Geometry and mathematics structures in computer science
	Optimizations
Theoretical computer science	
	Theoretical computer science
Software and data engineering	
x	Software engineering
x	Software development
	Web engineering
	Database systems

	Analysis and processing of large data sets
Software systems	
	System programming
	Reliable systems
x	Performance systems
Mathematical linguistic	
	Computer and formal linguistic
	Statistical methods and machine learning in computer linguistic
Artificial Intelligence	
	Intelligent agents
	Machine learning
	Robotics
Computer graphics and computer games development	
	Computer graphics
	Computer games development