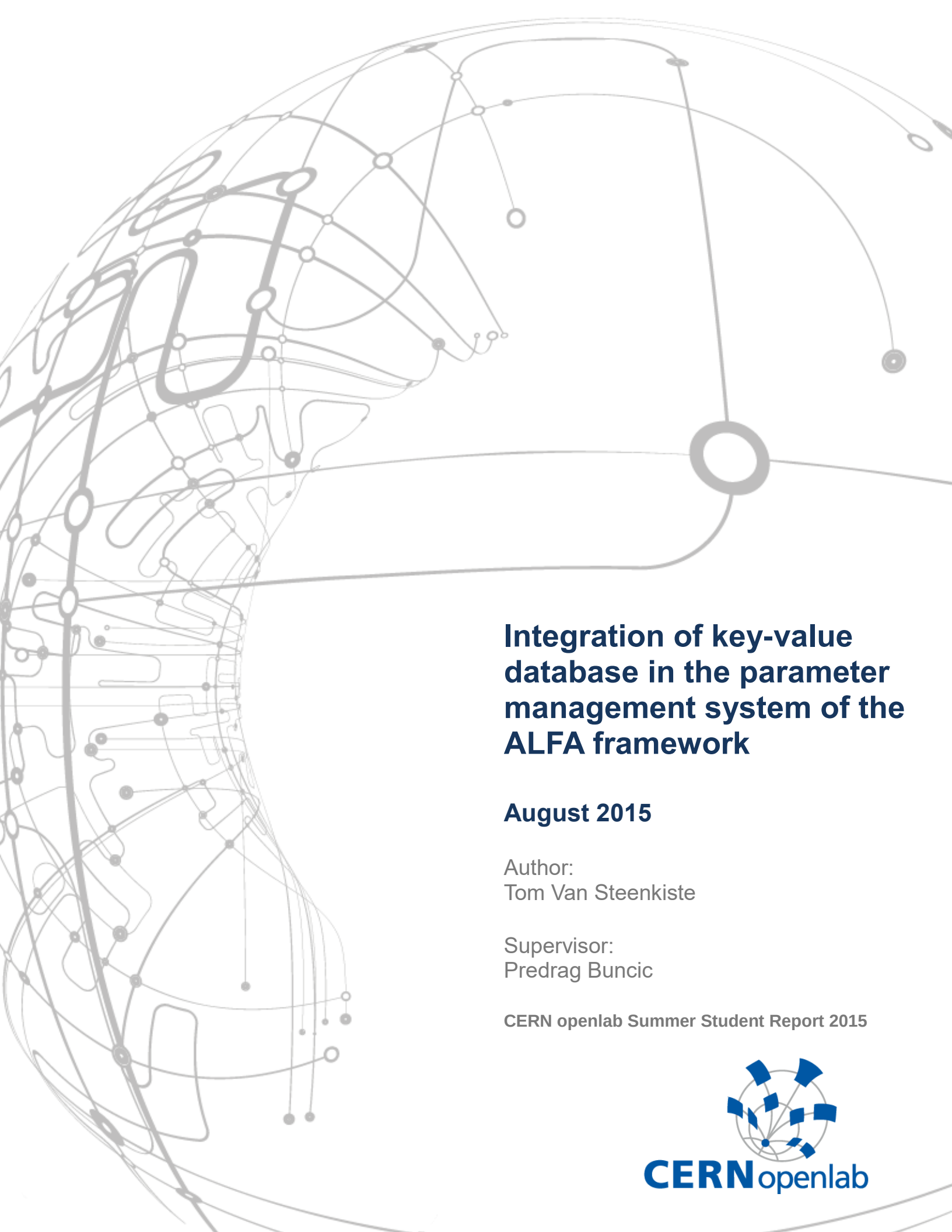

Integration of key-value database in the parameter management system of the ALFA framework

August 2015

Author:
Tom Van Steenkiste

Supervisor:
Predrag Buncic

CERN openlab Summer Student Report 2015

CERN openlab

Project Specification

The Parameter classes of the ALICE/FAIR (ALFA) software framework contain and manage all the numerical information needed to process the data. In order to analyse the raw or simulated data, several numerical parameters are needed, such as, calibration/digitization parameters or geometry positions of detectors. One common characteristic to most of these parameters is that they will go through several different versions corresponding, for example, to changes in the detectors' definition or any other condition. This makes it necessary to have a parameter repository with a well-defined versioning system. The runtime database (Parameter manager in ALFA) is such a repository. It knows about all parameter containers needed for the actual analysis. The containers can be initialized automatically from one or more inputs, and written out to one output. Possible inputs/output mechanisms are ROOT files or ASCII files.

In this project a key-value database will be investigated as an optional IO for the runtime database.

Abstract

In this project, several key-value databases are compared for their performance to implement them into the ALFA framework. As the research shows that a RAMCloud key-value database is not suited for this project, a deep research is done into Riak instead. The case-study compares read vs write latency, optimal simulation sample size, different storage backends, object size influence, cluster size influence, different consistency settings and there performance impact, the impact of adding security, the overhead of using a Java tool and the availability and fault tolerance of Riak. All these measurements are discussed taking into account certain difficulties of simulating a key value database. This gives us a document showing the impact of certain design decisions in implementing a key-value store and will allow developers for future projects to easily ascertain how changes in the database will impact performance and how to easily use a tool developed during this project to make exact measurements.

Table of Contents

1 Introduction

The ALFA framework is used to analyse raw or simulated data. When reconstructing events from this data, the conditions and parameters of the events are important. A distinction is made between two kinds of parameters. There are fast changing parameters which are stored with the data itself and slower changing parameters which can be stored in a more efficient way. This project investigates the use of a key-value database to store these parameters.

First, the requirements for a key-value database in the ALFA framework are researched. Then, a general study into several key-value databases is done which focusses on the requirements of the implementation in the ALFA framework. Next, a performance testing framework is developed to equally tests all key-value databases according to the specific use case of the ALFA framework. This performance testing framework is then used to test the key-value databases and several important statistics are derived from these tests. Finally, a general overview of the results of this project is given.

2 Database requirements

The first step of this project is investigating the key-value database requirements which will allow us to focus on the important features. A first important aspect of the key-value database is that the ALFA framework should be able to interface with it. This means that a C/C++ interface should be available or it should be possible to develop one.

Next, we investigate the required object sizes to be stored. This can be found in the ALICE requirement documents. [1] [2] From these documents we can derive that object sizes vary between a few kB and 50MB. To test this range of object sizes, the following sizes should be taken into account: 1kB, 10kB, 100kB, 1MB, 10MB and 50MB.

The access pattern is also important for the simulations. For this, we are particularly interested in the number of reads versus the number of writes. The current statistics for the OCDB show 200 files read per run and 60 files written per run or 200MB read per run and 30MB written per run. [2] These statistics give us a 3.33 reads per write ratio when looking at the number of files or a 6.66 reads per write ratio when looking at file sizes. As the final access pattern in the simulations, we take the 3.33 reads per write ratio. The most important reason is that writes usually take longer than reads in key-value databases and thus 3.33 reads per write represents the worst case scenario.

For the simulations in this research, we will thus investigate the performance of key-value databases with 3.333 reads per write and object sizes of 1kB, 10kB, 100kB, 1MB, 10MB and 50MB. As no exact information is available about the precise access pattern, such as when reads happen and when writes happen, a random access pattern converging to 3.333 reads per write will be used. Since there is also no information on the specific distribution of the value sizes, each size will be tested separately.

3 Key value databases

Multiple key-value databases are examined for their properties and performance for the use in the ALFA parameter framework. The two most important candidates are RAMCloud and Riak. The parameter framework only requires very basic functionality such as PUT and GET as all other functionality such as versioning is handled by the parameter manager by generating different keys. This enables us to focus on performance and robustness instead of extended functionality sets.

3.1 RAMCloud

RAMCloud is a key-value database designed at Stanford University with an explicit focus on performance. It offers low latency access for small objects and scalability in the amount of storage nodes. More information on RAMCloud itself can be found on the project website. [3] Due to the focus on performance, it seems like an ideal candidate for this project. However, the documentation only mentions the high performance for small objects such as zero-byte objects. As mentioned in section 2 the database should be able to store larger objects up to 50MB.

The maximum object size in RAMCloud is 1MB which is less than the size of certain parameters currently in use as specified in the database requirements section. [4] A possible solution to the restricted size of objects in RAMCloud is storing them by splitting them up in chunks and merging them again when requested. To test possibilities of using RAMCloud with chunked objects, the performance near its object limit of 1MB should be tested to see the impact of larger objects.

RAMCloud is written in C++. To integrate it into the ALFA framework, only the RAMCloud header has to be included and all functionality of RAMCloud is available. For library requirements, please see the official RAMCloud documentation. [3]

3.2 Riak

Riak is a key-value store designed by Basho. It focusses on high availability, scalability and fault tolerance and offers a lot of extra functionality. More information on Riak itself can be found on the Basho website and the Riak documentation site. [5] [6] Riak differs from RAMCloud by allowing users to configure it’s consistency configuration and other settings and allowing them to store larger objects.

To integrate Riak into the ALFA framework, two options are considered. First, a C/C++ client is developed. However, as support from Basho for the C/C++ client is limited, a Java client is also considered.

3.2.1.1 C++ Client

For integrating a Riak database with the ALFA framework, it is preferred to use a C/C++ library for interfacing. Riak does not officially support a client library in C or C++. However, several libraries are available on their website. An example of this is the riak-c-client library. [7] This library is not ready for production use. When contacting Basho concerning several problems with the library, we were recommended to not use it as it will not be supported in the future. However, a basic test implementation with the riak-c-client library has been made.

Other suggested libraries are the riack library or riack-cpp library. [8] [9] However, these libraries are either in a very early stage or have been abandoned. As good support is needed for the libraries, these options are not a good choice. Because no C/C++ libraries with guaranteed future support are found no C/C++ implementation is used.

3.2.1.2 Java Client

As no C/C++ library with sufficient support is available, the next option is to use a library in an officially supported language and interface it with the ALFA framework in C++. An example of one of these libraries is riak-java-client. [10] This library is chosen as it offers support for future implementations.

The implementation of the interface can be found in the RiakJavaC project. [11] The interface consist of a Java project which uses the official riak-java-client to communicate with Riak. To allow the ALFA framework, which is written in C/C++, to communicate with the RiakJavaC project, the Java project accepts request-messages form any programming language using Google Protocol Buffers. These protocol buffers are send over a low overhead protocol called ZeroMQ which was chosen as it is already used in communications of the ALFA framework. [12]

The proto files for the request-messages are also contained in the RiakJavaC project. Request-messages can contain 4 fields of which one, the command string, is required. It indicates what is expected with the data in the other fields. Programs can send the PUT, GET and ERROR commands, indicating the required operation or an error. The reply commands are OK and ERROR indicating a successful or a failed operation. The three optional fields are the key, value or error strings. In the same project, an example implementation for a C++ program, communicating to the Java client via these messages is shown. This can be used as a template to interface any C/C++ application with the Java client.

The Java project consists of three main parts: the messaging package, the riak package and the client package. The client package is the main package and starts up a MessageParser, MessageListener and a RiakConnection. It also parses user commands from the terminal via the CommandParser. The command parsing feature is used for debugging purposes and should not be used for implementations. The MessageListener makes a connection towards a load balancer which distributes the load of the requests on the different JavaRiakClient instances. It's only task is sending and receiving messages. When a message is received, it is forwarded to the MessageParser to be interpreted. The MessageParser performs the command received in the request on the RiakConnection object and returns the resulting data or an error message. Communication between these entities can be seen in Figure 1. Please see

Appendix A – Using RiakJavaClient for information on how to use this program. To get more information on the implementation details of the Java project, please generate the JavaDoc documentation.

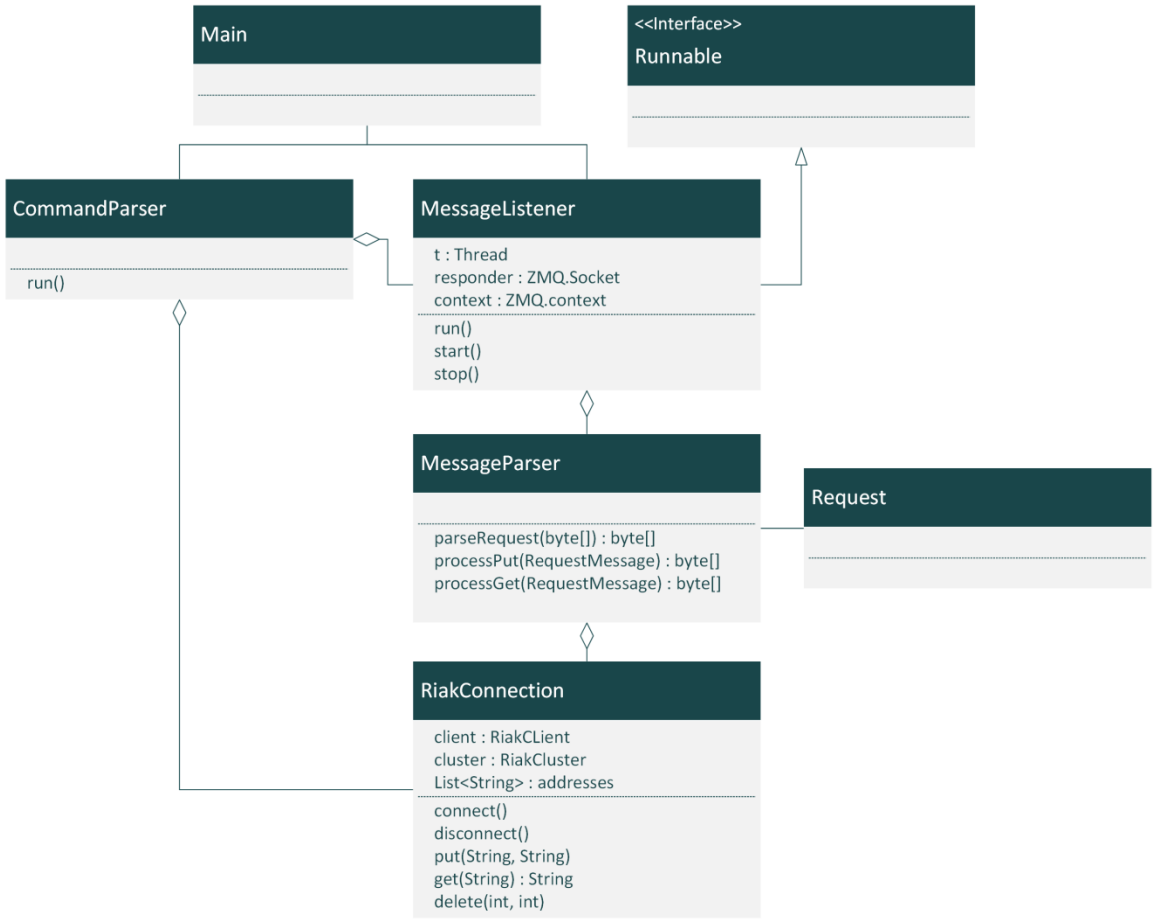

Figure 1 - UML Class Diagram RiakJavaClient

To allow for easily extending the Riak Cluster without having to update the application configuration, an extra node is added called the load balancer or broker. This node takes in requests from the applications connecting to it and distributes them to RiakJavaClient instances who connect to the load balancer as well. Currently, the load balancer is implemented according to the standard example implementation of the Router-Dealer pattern in ZeroMQ as seen in Figure 2. [13] The application in this figure can be any application written in a language supporting Google Protocol Buffers and having a ZeroMQ implementation. Currently, the load balancer uses the standard algorithms provided by the ZeroMQ framework. A future research aspect might be into more advanced load balancing algorithms tailored to the specific use case of this project.

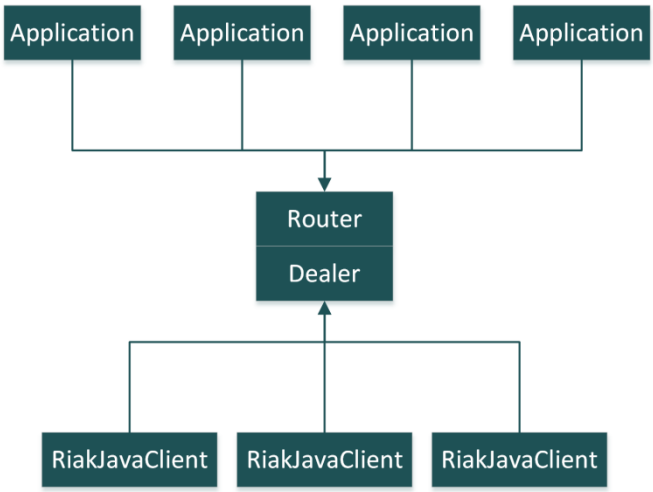

Figure 2 - Router Dealer pattern in ZeroMQ

By using the load balancer pattern, connecting applications only have to know how to reach the load balancer. They don't have to be aware about the backend implementation of Riak and how many Java clients are online. The Java clients themselves are also easily configured as they only need to connect to the load balancer and the Riak cluster.

4 Database performance framework

One of the main focusses of this project is testing the performance of key-value databases for use in the ALFA framework. Many testing tools for databases already exist as most developers write a benchmarking tool for their own database. However, for this research a universal tool is needed to test each database under the same conditions which can be adjusted to the required parameters for the framework. At the same time, the entire communication pipeline is tested as well, instead of just the key-value database itself. Using the same tool with the same configuration to test various key-value databases will also give a clear view on how they differ in performance for specific use cases.

4.1 Design

A benchmarking framework for any general key-value database called KeyValueClusterPerf is developed to test the various key-value databases. The framework is designed to be easily configurable and was developed to cope with simulations of thousands of nodes at the same time. If extensions are needed for a certain use case, they are easily implemented. For testing a new key-value database one only has to write an interface class to that database with basic PUT and GET operations.

The connections between the various classes of the framework is shown in Figure 3. The main entry point to the program is the KeyValueClusterPerf class. This class parses the command line arguments and launches either a SimulationWorker or a SimulationController depending on the arguments. The SimulationController class contains all functionality to organise tasks across the other instances of KeyValueClusterPerf that run SimulationWorkers, which receive those tasks and perform the actual simulation, after which they report back the results. The SimulationController is executed on the user workstation and contacts the workers via a hostfile

specifying the different workers available. The controller expects an instance of KeyValueClusterPerf in worker mode to be running on each of the specified hosts.

If a large number of SimulationWorkers are to be contacted, care should be taken that the workstation configuration is adequate to handle the task. Especially the file limit and thread limit can be of concern as a lot of ports are opened to connect to all the workers. If there are a lot of simulations rapidly executing one after the other, the node may run out of file handlers because of a too large linger time on TCP connections.

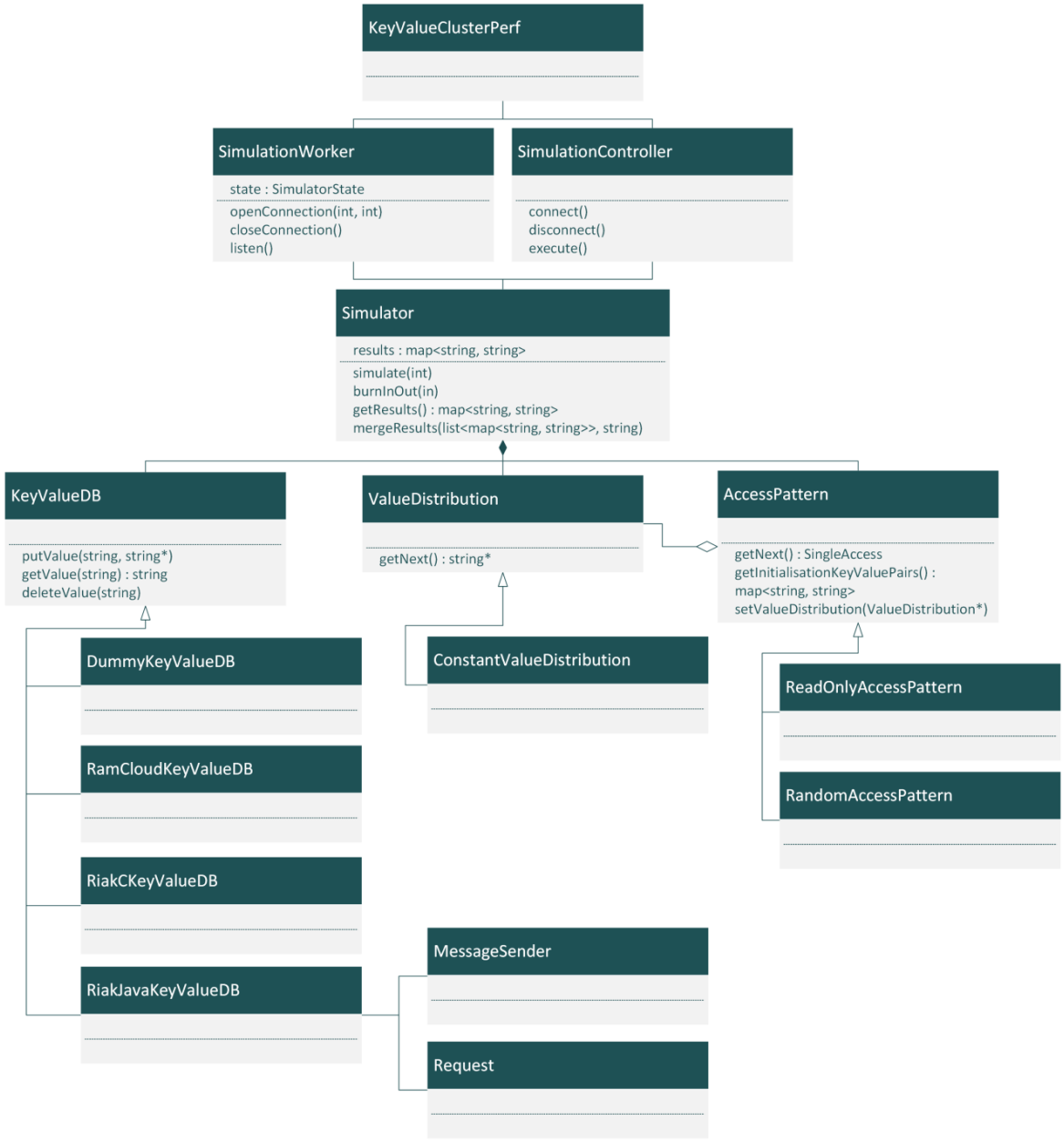

Figure 3 – UML Class diagram KeyValueClusterPerf

During the simulation, the worker and controller go through several states. The first step is the initialisation phase. During this phase, all necessary keys are written to the key value store. This prevents requests to non-existent keys during the simulation which could impact performance when the key-value database returns an error instead of the expected value. This could impact the correctness of the results. This phase can be skipped via an extra command line argument. It is recommended that only one worker node performs this phase as more workers would result in unnecessary delay. However, when workers are given their own key-space, they should each initialise this space.

After initialising all the possible keys via a worker node and receiving a success reply, the controller sends the go-command to all worker nodes. The nodes then start a burn-in period performing requests on the database according to the actual supplied access pattern and value size distribution. This burn-in period is required to counter some small synchronisation issues when thousands of nodes are fired up at the same time. After the burn-in period, the simulation starts following the exact same access pattern but now timing information is also stored. Finally a burn-out period allows straggling nodes to still get their timing information under the same load. The burn-in, simulation and burn-out functionality is defined in the Simulator class.

When all nodes have finished the burn-out, the controller requests their timing data. As this data can be a lot larger than the commands used to control the worker nodes, a separate port which allows larger data chunks to be send is opened to send the data through. The controller then merges the data using the Simulator class. When all of the data has been received, the controller sends the exit command to all worker nodes to shut them down safely. A possible extension to this mechanism could be to send a reset command to the worker nodes so they can immediately be reused in a next simulation.

Users can specify their own access pattern, value size distribution and key-value database. The framework is designed to only use the functionality supplied in the AccessPattern, ValueDistribution and KeyValueDB abstract classes so only these functionalities have to be fulfilled by the user specified access patterns, value size distributions or key-value databases. However, some custom classes could require their own configuration parameters. To solve this, a special ConfigurationManager class is designed to read in configurations form files and supply the configuration as a key-value map to the constructor of each access pattern, value size distribution and key-value database.

All code for this framework, including doxygen documentation can be found on GitHub. [12] For information on how to use the tool, please see Appendix B – Using KeyValueClusterPerf.

4.2 Configuration

The KeyValueClusterPerf framework uses configuration files to correctly configure the key-value database, access pattern and value size distribution. In this section, information on how to configure the standard access patterns and value distributions is given. Note that in all configuration files, comments can be added by placing a # before the line.

4.2.1 Database

To select a specific database, write a key “databaseType” with a specific value from Table 1 in the configuration file. Then, add extra configuration lines for that specific database.

Key-value database class	Key-value database type definition
RamCloudKeyValueDB	RamCloud
RiakCKeyValueDB	RiakC
RiakJavaKeyValueDB	RiakJava
DummyKeyValueDB	Dummy

Table 1 - Key-value database type

To configure a RamCloudKeyValueDB, the communicationprotocol needs to be specified as well as the hostname of the coordinator and the port at which it can be reached. These values are specified using the “communicationProtocol”, “coordinatorHost” and “coordinatorPort” keys.

For a RiakC database no extra configuration is currently used. However there are some values in the code that can be exported to a configuration file.

To configure a RiakJava database, one can specify a “security” key to indicate which security protocol, sha1 or VMAC, to use. This key is not mandatory. An example implementation of this can be seen in Listing 1 where a RiakJava database is configured to use VMAC as a security protocol.

The dummy database accesses no real database and uses no configuration files.

```
databaseType=RiakJava
security=vmac
```

Listing 1 – Database configuration file example

4.2.2 Access pattern

To select a specific access pattern, write a key “accessPatternType” with a specific value from Table 2 in the configuration file. Then, add extra configuration lines for that specific access pattern. For the read-only and write-only access patterns, only the minimum and the maximum key should be specified. The access pattern will then initialise all these keys and during simulation will pick a key in the specified range. This can be useful when also testing large key sizes. The minimum key is specified with the “minKey” value and the maximum key with the “maxKey” value.

Access pattern class	Access pattern type definition
RandomAccessPattern	Random
ReadOnlyAccessPattern	ReadOnly
WriteOnlyAccessPattern	WriteOnly

Table 2 - Access pattern types

To configure a random access pattern, also a read-to-write ratio should be specified with the key "readWriteRatio". This is a number in double format specifying how many reads should be performed on average to every write. An example implementation for an access pattern configuration file can be seen in Listing 2 where a random access pattern is defined with a key range between 1 and 1000 and where there are twice as many reads as writes.

```
accessPatternType=Random
minKey=1
maxKey=1000
readWriteRatio=2.0
```

Listing 2 - Access pattern configuration file example

4.2.3 Value distribution

To configure a value size distribution, write a key "valueDistributionType" with as value the type, as specified in Table 3, in the configuration file. Then, add extra configuration lines as with the access pattern configuration files. To configure a constant value distribution, only the size in bytes is needed. An example of this can be seen in Listing 3 where a constant value distribution is defined with a value size of 1kB.

Value size distribution class	Value size distribution type definition
ConstantValueDistribution	Constant

Table 3 – Value size distribution types

```
valueDistributionType=constant
size=1000
```

Listing 3 - Value size distribution configuration file example

4.3 Extensions

The performance measurement framework is designed to be easily extensible to fit the users' needs. One can add a new key-value database type, a new access pattern type or a new value distribution in a straightforward fashion. This section will give a brief overview on how to create a new extension for all three examples.

4.3.1 Database

A new database type can be added by creating a new class that inherits from the abstract KeyValueDB class. To keep the same consistency throughout the code, a constructor has to be made that has a map<string, string> as input parameter containing the configuration for this database type.

When the functionality for this specific database has been implemented in the PUT and GET functions, users only have to add another 'else if'-option in the constructor function of the Simulator class to load the new database. For an example implementation, please see the DummyKeyValueDB in the code. This KeyValueDB has been designed to test the performance of the framework itself.

4.3.2 Access pattern

To add a new access pattern type, a new class should be implemented that inherits from the abstract AccessPattern class. As with the KeyValueDB class, it is recommended to implement a constructor with a map<string, string> as input with configuration information.

As a last step, the access pattern has to be added as an 'else if'-option in the constructor of the Simulator class.

4.3.3 Value distribution

A new value size distribution type can be added in exactly the same fashion as a new database or access pattern type by inheriting from the abstract ValueDistribution class, adding a correct constructor and adding the 'else if'-option in the Simulator class.

5 Case-Study: RAMCloud

RAMCloud is the initial candidate of this research. All implementations and integrations for RAMCloud have been made. As mentioned in section 3.1, RAMCloud is not able to store large objects. However, if the performance near the object limit is sufficient, this can be solved by chunking the objects.

The RAMCloud documentation suggests read times of 5µs and small write times on the order of 15µs. [3] It should be noted that these access times are for Infiniband connections and that the test cluster available for this case-study is only equipped with 1Gbps connections.

The key focus of RAMCloud is thus around the 1MB limit. However, during initial testing the performance of RAMCloud degraded very rapidly when reaching this limit leading to unresponsiveness. As the requirements for the project require object sizes up to 50MB no further research into RAMCloud has been done.

6 Case-Study: Riak

Riak is the second candidate in this research. First, a study is done if it is necessary to distinguish between reads and writes in the analysis of the data. Then, the ideal sample size for simulation is analysed. After that, a comparison is made of different backends for Riak. Then, the impact of object sizes is analysed. A next research is testing the impact of the cluster size on the performance. Next, we study the impact of the configuration of Riak after which we study the impact of adding security to the communication. Then, we research the impact of changing the amount of Java nodes and do a literature study on availability and fault tolerance or Riak. Finally, we discuss some difficulties with simulating key-value databases.

6.1 Read vs Write latency

To know if it is important to distinguish between read and write operations in the analysis of the results, a test experiment is set up to see the difference in write duration and read duration when both happen at the same time. In Table 4, the results of an experiment can be seen where 1 host constantly reads data and 1 host constantly writes data. As the table shows, the difference in duration is very small and not consistent. Sometimes reads are more efficient and sometimes writes are more efficient. From this we can ascertain that the costs of the operations are amortized across all queries when the database is under load. It is thus sufficient to do the tests with a 3.33 reads to write ratio and afterwards process all data in the same analysis.

Size	Read duration	Write duration	% Difference
1kB	0.003456	0.003455	-0.033927
10kB	0.003673	0.003675	0.066285
100kB	0.008708	0.008708	0.005512
1MB	0.085394	0.085384	-0.010775
10MB	0.936064	0.935816	-0.026501

Table 4 - Read vs Write latency

6.2 Sample Size

As performing extensive simulations on a key-value store can take a lot of time, the next step is investigating how large the sample set should be. For 3 different sample sets, a series of 5 simulations are performed. From these data sets, we extract the values we are interested in, the latency and throughput, and compare the standard deviation of these values across the 5 series. These results are shown in Figure 4 and Figure 5. From the graphs we can see that performing 100 simulation iterations is not sufficient, however there is not a lot of difference between the 250 simulations and the 1000 simulations.

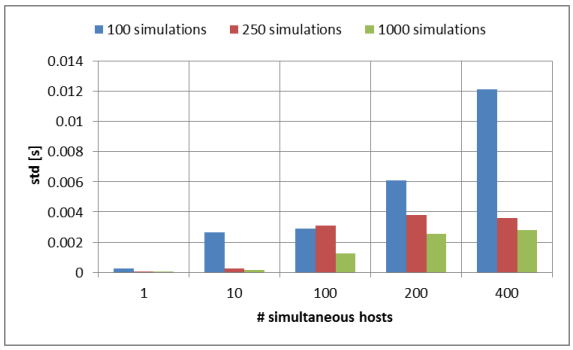

Figure 4 - Standard deviation on latency after 5 series

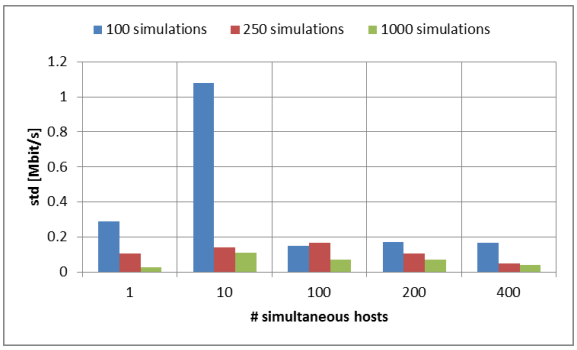

Figure 5 - Standard deviation on throughput after 5 series

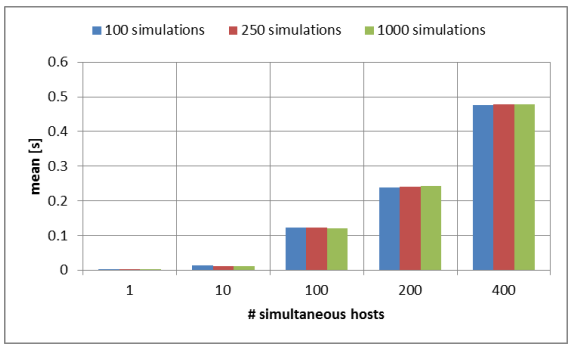

Figure 6 - Mean of latency after 5 series

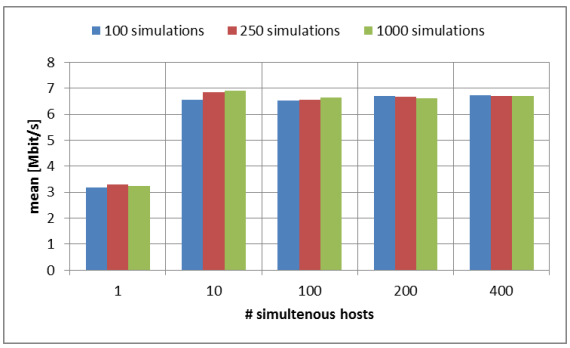

Figure 7 - Mean of throughput after 5 series

When we take a look at the averaged results across the 5 series, an interesting result appears in Figure 6 and Figure 7. The averages across the 5 series are really close to each other, regardless of the amount of performed simulations. It is thus more efficient to do multiple smaller simulation sets than to do 1 big simulation set. In the following simulations, we always perform 250 simulations.

6.3 Storage backend: Bitcask vs LevelDB

Riak can be equipped with different storage backends. The storage backend takes care of persisting the database to hard disks while Riak takes care of distributing the data across the different nodes. Originally, Riak is shipped with Bitcask enabled. However, the storage backend can be easily exchanged with other preinstalled storage backends such as LevelDB by changing a value in the configuration of Riak.

During the tests, the simulations often fail due to the Riak nodes being out of memory. The source of this problem is the way Bitcask persists the data to hard drives using a log structure. [13] This prevents us from performing all tests using a Bitcask backend because of which, all tests will be performed on a LevelDB backend. However, we do want some indication on the performance implications of using LevelDB instead of Bitcask. We run tests with one Riak node and the different backends. The results can be seen in Figure 8 and Figure 9 for the latency and Figure 10 and Figure 11 for the throughput.

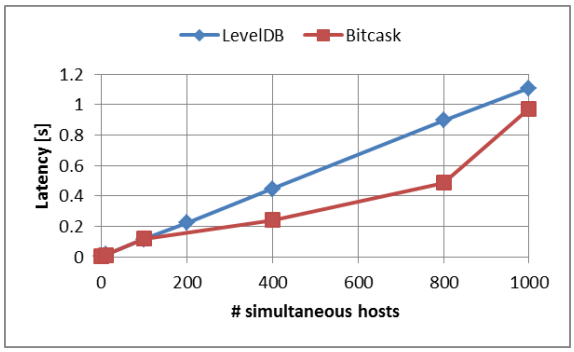

Figure 8 - Latency using 1 Riak node with 1KB size

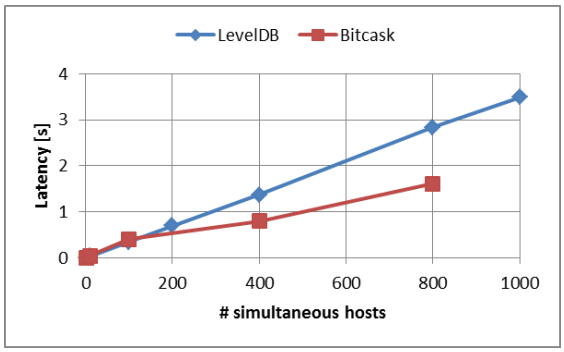

Figure 9 - Latency using 1 Riak node with 100KB size

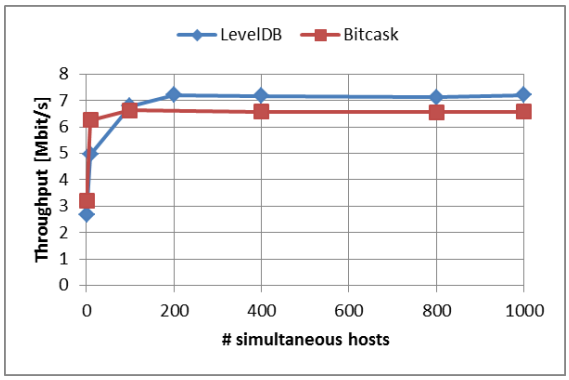

Figure 10 - Throughput using 1 Riak node with 1 KB size

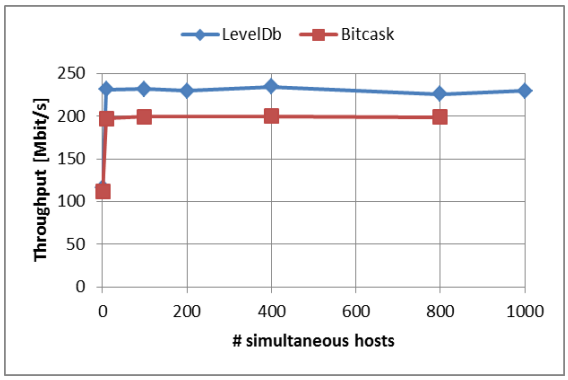

Figure 11 - Throughput using 1 Riak node with 100KB size

The Bitcask results in the graphs are incomplete as performing these simulations would results in the Riak node going down due to lack of memory. This shows the need for a different backend. As can be seen from the graphs, the performance for latency of Bitcask is better than for LevelDb. However, LevelDB is able to reach a higher throughput.

The main performance measure of interest is latency. During the performance tests, a lot of data is written to the database. This is a problem for the performance tests as the nodes cannot store that amount of data. Even when deleting the data afterwards, the storage of Bitcask keeps growing. For this reason, LevelDB is the ideal candidate for the performance tests. The final implementation can, however, use Bitcask as all data needs to be kept anyway and will never be updated or deleted. For all following tests, LevelDB will be used as backend.

6.4 Object size

Next we investigate the influence of the object size on the latency and the throughput. The latency of the requests can be seen in Figure 12 and Figure 13 and the throughput can be seen in Figure 14 and Figure 15. Note that both parts contain the same 100kB entry. The graphs have just been split up to make the smaller measurements clearer.

We see the same trend for all object sizes, however the increase in latency is more severe when object sizes grow. We can also see that the throughput is fairly constant throughout the increase of the number of simultaneous hosts. This means that for a certain object size, the throughput is fixed.

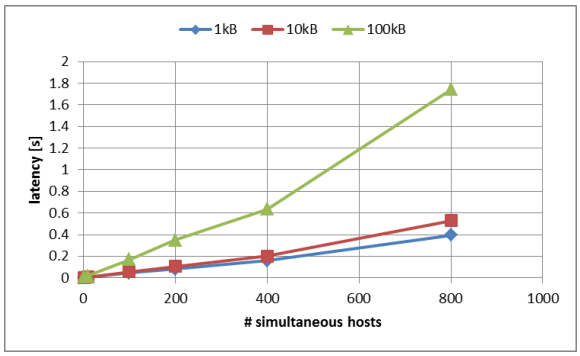

Figure 12 - Latency for objects of size 1kB to 100kB

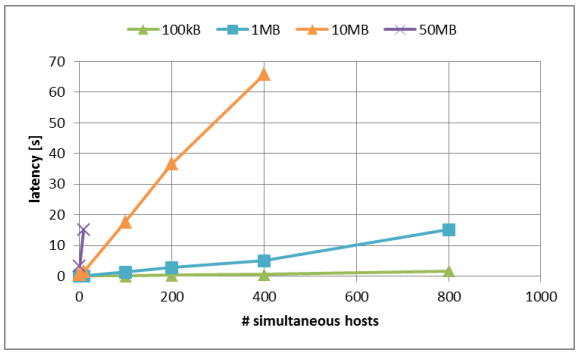

Figure 13 - Latency for objects of size 100kB to 50MB

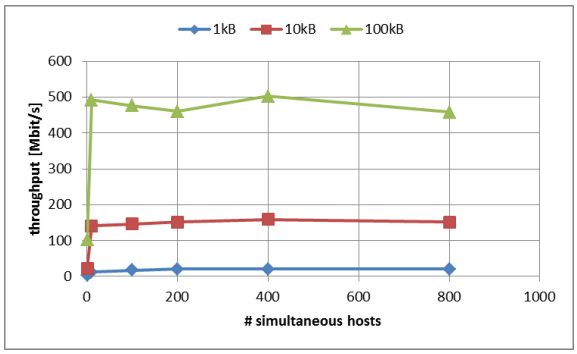

Figure 14 - Throughput for objects of size 1kB to 100kB

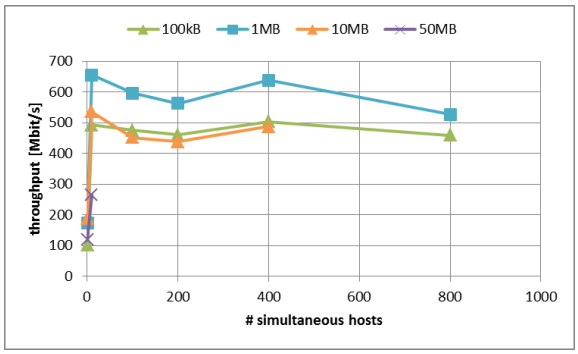

Figure 15 - Throughput for objects of size 100kB to 50MB

We see that for large objects, there is a large latency. However, these graphs have to be correctly interpreted. They have been made using a constant value size. In a real application, not all requests will be of that size. As the distribution of the value sizes is not known, we only simulated a fixed value size to get an insight into the impact of object sizes on the database. To get a clearer view, the distribution of value sizes should be measured and used in an extra simulation. The measurements do give us an overview of how the database reacts to different object sizes.

6.5 Cluster Size

To get a view of the trend when increasing the Riak cluster size, 4 different cluster sizes are tested: 1 node, 2 nodes, 4 nodes and 8 nodes. Note that in all these arrangements, the amount of Java clients scales the same as the amount of Riak nodes. In Figure 16 and Figure 17, the latency is shown for object sizes of 1kB and 1MB respectively. In Figure 18 and Figure 19, the

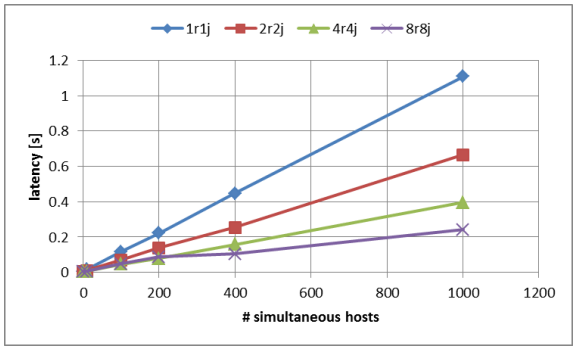

Figure 16 - Latency using 1kB sized objects

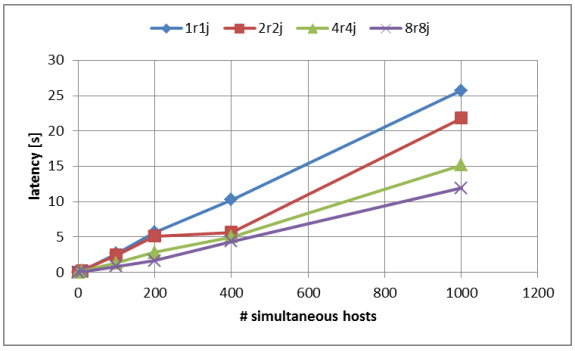

Figure 17 - Latency using 1MB sized objets

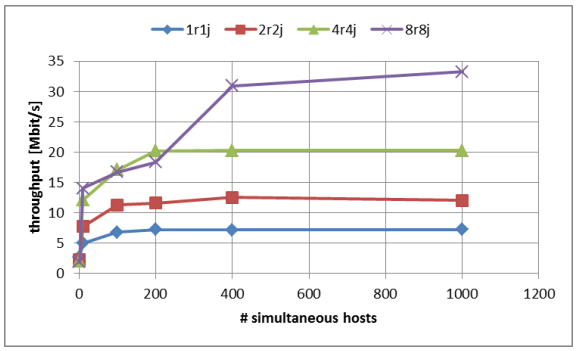

Figure 18 - Throughput using 1kB sized objects

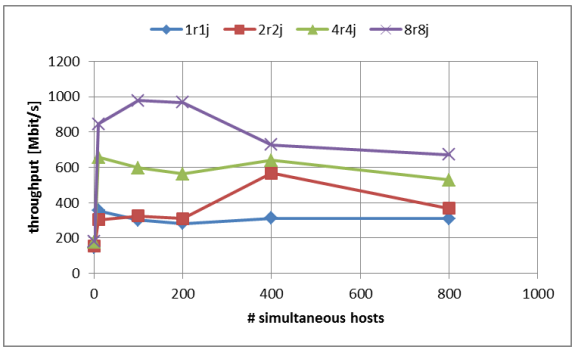

Figure 19 - Throughput using 1MB sized objects

throughput is shown for object sizes of 1kB and 1MB respectively. We can see that increasing the cluster size improves the performance in terms of latency and in terms of throughput. We can thus derive from this that it's advantageous to have a larger cluster size. In section 6.9, we also discuss the impact of increasing the cluster size on the availability.

Note that for 200 simultaneous hosts, with 8 Riak nodes and 8 Java clients, we reached the link bandwidth of 971Mbit/s. When more hosts are added, the performance drops significantly as the link was already saturated and all requests will now be further delayed.

6.6 Riak configuration

Riak is highly configurable and allows it's users to select certain consistency parameters. Two of those parameters are the amount of read acknowledgements and the amount of write acknowledgements. In the default settings, 3 nodes reply to a read and 3 nodes reply to a write. This provides extra consistency settings as some nodes may already have an updated version of some data while others don't.

In Figure 20 and Figure 21, the latency for objects of size 1kB and 1MB respectively can be seen. In Figure 22 and Figure 23 the throughput for these object sizes can be seen.

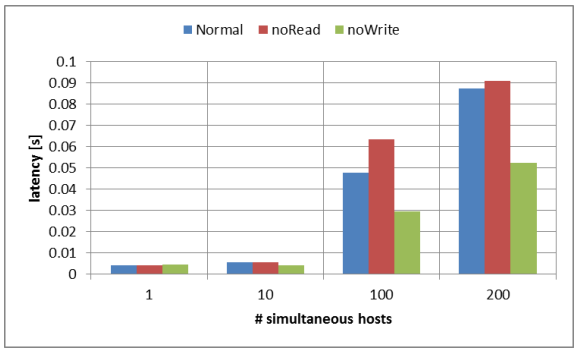

Figure 20 - Latency using 1kB sized objects

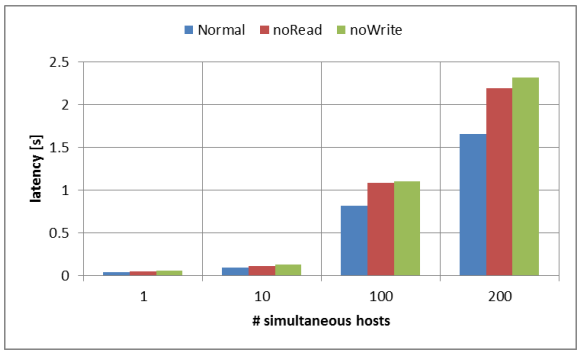

Figure 21 - Latency using 1MB sized objects

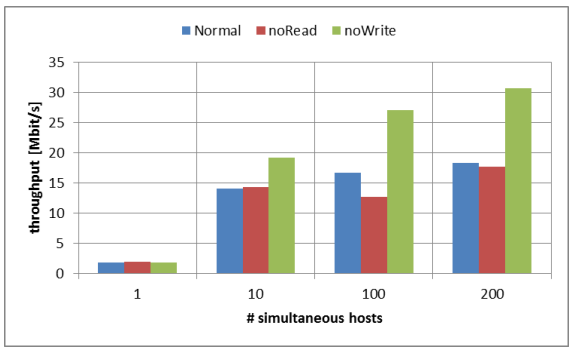

Figure 22 - Throughput using 1kB sized objects

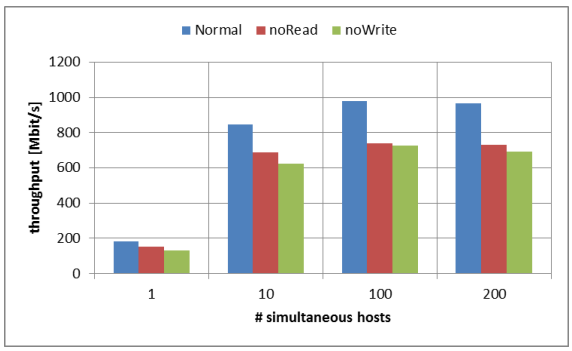

Figure 23 - Throughput using 1MB sized objects

As expected, the latency goes down when we only ask for 1 node to acknowledge the write (no write consistency) for small objects. The throughput also goes up in this case. However, when we look at the data for when only 1 node should reply to a read (no read consistency), we see that the latency increases. There are two possible explanations for this. The first is that there's an added overhead in the Riak cluster itself to verify that the latest data is read. The second is an error happened during the measurements. A cause for this is explained in section 6.10.

When we look at larger objects however, both for no read acknowledgments as for no write acknowledgements, the latency goes up and the throughput goes down. This can again be due to an error in the measurement or due to background processes of Riak. More research has to be done in this section.

6.7 Security via message authentication

Another aspect of the project is looking into the effect on performance by adding security features to the communication. The only required security feature is message authentication. The Riak Java client has to be sure that the message came from our application and that it has not been tampered with. To achieve this, two possibilities are investigated. First, an HMAC is appended to the communication between the ALFA framework and the broker. We can safely assume that the communication between the broker and the Riak cluster and the API of the Riak cluster itself is sufficiently secured by means of firewalls. The other possibility is using a VMAC.

As we are only interested in the performance of this operation, we do not consider the problem of distributing the keys or any other data needed for the authentication. The only aspect we consider is the duration of calculating the authentication over the full message. The calculated MAC is also not appended to the message as the impact on the final message size is negligible. As the calculated string is never used, some tricks are needed to prevent the compiler from optimising away the calculations. The end result of the calculation is stored as a publicly accessible variable to the class to prevent any optimisations.

For calculating the HMAC, a SHA1 is used. The communication between the application and the broker requires weak collision resistance, also called second preimage resistance. As no strong collision resistance is needed, a SHA1 over the message and a private key suffices to fulfil the security requirement. In the KeyValueClusterPerf framework it is implemented via the SHA1-library by Steve Reid, Bruce Guenter and Volker Grabsch. In the broker, basic Java functionality is used to calculate it. The broker can be started in security mode by adding the 'secure' command line argument. The KeyValueClusterPerf framework can be started in security mode by adding a security specification in the database configuration file for the RiakJavaDB.

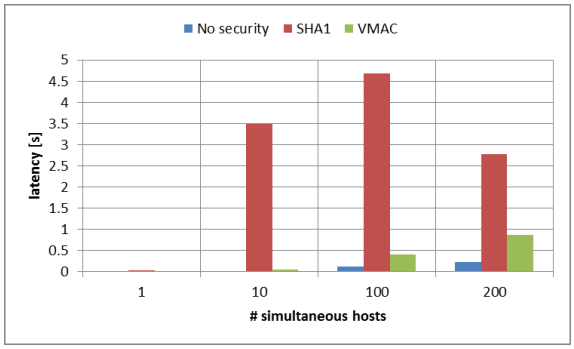

Figure 24 - Latency using 1kB sized objects

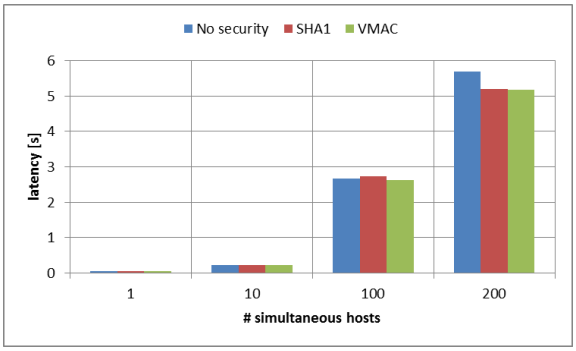

Figure 25 - Latency using 1MB sized objects

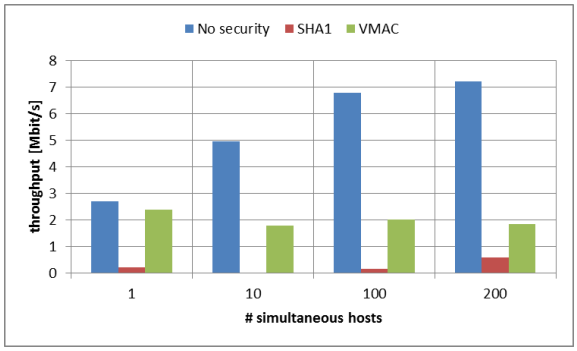

Figure 26 - Throughput using 1kB sized objects

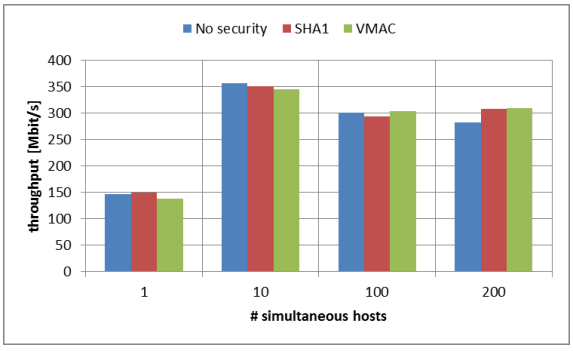

Figure 27 - Throughput using 1MB sized objects

An HMAC via SHA1 is a very common implementation. However, another interesting implementation is using a VMAC which has been optimised for speed. For the KeyValueClusterPerf framework, the VMAC implementation by Ted Krovetz is used. This implementation uses assembly code and memory alignment to get the best possible performance. For the Java broker however, there is no existing implementation. Because of this, only the difference in performance at the application side, in comparison to SHA1 can be measured.

Figure 24 and Figure 25 show the difference in latency and Figure 26 and Figure 27 show the difference in throughput between no security, SHA1-authentication and VMAC-authentication. We can clearly see that for small objects, that the SHA1 implementation increases the latency a lot. In comparison the VMAC implementation doesn't increase the latency that much. However, the VMAC implementation still multiplies the latency by a significant amount. When looking at the throughput, the difference between the VMAC and the no security is very clear. The drop in latency between 100 simultaneous hosts and 200 simultaneous hosts can be explained by the fact that under normal circumstances, the benchmark is highly IO-bound. However, with the addition of the security features, the benchmark also uses a lot of CPU power.

When looking at the performance for larger objects, we see that the latency actually decreases. This might be due to an measurement error although this result has been repeated. An explanation could be that with larger objects, the hosts try to contact the database less often because they are busy calculating the security codes and thus actually reduce the load on the database. More research has to be done into this aspect.

6.8 Amount of Java clients

During all previous tests, when we scaled the cluster size, we also scaled the amount of Java clients for the cluster equally. In this section we investigate the actual influence of scaling the Java client.

In figure Figure 28 and Figure 29, the latency for three different setups can be seen for objects sizes of 1kB and 1MB respectively. In Figure 30 and Figure 31, the throughput for these same setups can be seen. The three setups under study are a cluster with 8 Riak nodes and 8 Java clients, a cluster with 1 Riak node and 8 Java clients and a cluster with 8 Riak nodes and 1 Java client.

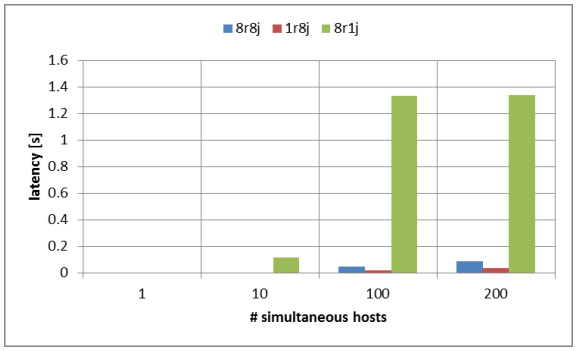

Figure 28 - Latency using 1kB sized objects

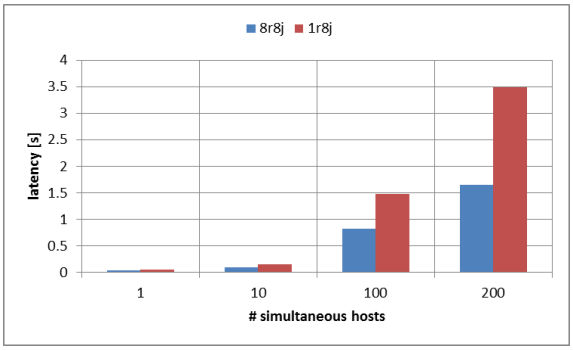

Figure 29 - Latency using 1MB sized objects

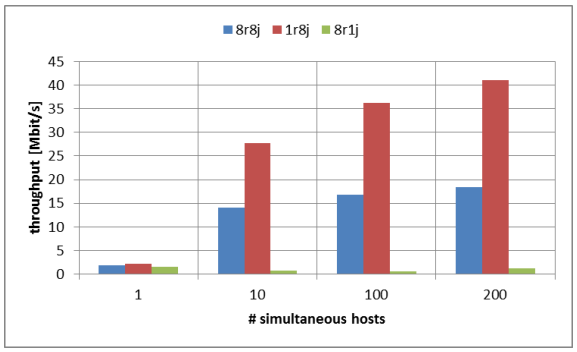

Figure 30 - Throughput using 1kB sized objects

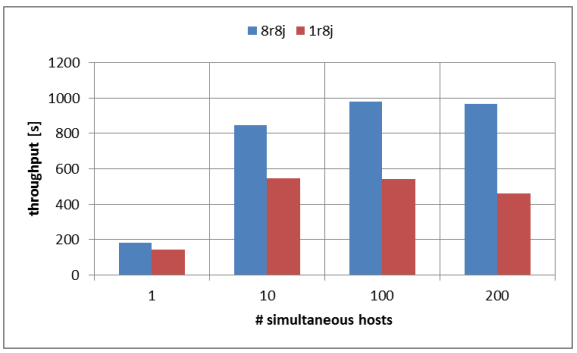

Figure 31 - Throughput using 1MB sized objects

When looking at the graphs, we can clearly see that having only 1 Java client has a significant detrimental impact on the performance. From this we can derive that the Java client is a bottleneck in the application. We can also see that for small objects, it's advantageous to have a smaller Riak cluster than usual. However, for larger objects, the larger Riak cluster has more advantages. Note that no measurements have been done for large objects and 8 Riak nodes and 1 Java clients as this would not supply any more relevant data and other measurements still had to be performed.

6.9 Availability and fault tolerance

No physical test have been done to investigate the availability and fault tolerance performance of Riak. However, a small investigation into this subject has been made.

Riak is highly configurable, and so is it's replication policy. By default, data is stored on 3 different vnodes. These vnodes can be distributed across multiple physical nodes or even across different datacentres in case of Riak Enterprise. However, this is only possible if the amount of Riak nodes is sufficient to store the amount of replicas.

From this we can learn that having more nodes increases the availability and fault tolerance. If more assurance is needed, the amount of replicas can also be changed to distribute it across more nodes.

There has been no investigation into the recovery performance of Riak after a node went down. However, a significant impact can be expected as background processes will try to redistribute all data in the cluster to guarantee its safety. More research has to be performed in this section.

6.10 Difficulties during simulation

In some sections, it was mentioned that not all data can be trusted and that there might be errors in the data. This is due to typical difficulties of benchmarking databases and in particular Riak. When testing the database, a clean database is preferred. However to clean it out each time after running only 1 test takes a significant amount of time and might fire up some background process which could impact performance longer than anticipated.

Another difficulty during simulation is that the database might start background processes without the knowledge of the tester. An example of this is a merging process which starts when the nodes use Bitcask as a backend and are almost running out of memory. Then, a process will

go through the log files to try to merge them and make them smaller. However, this process impacts performance significantly and could alter simulation duration results.

Other possible impacts on performance during simulation are other processes, not from the database, interfering with the processing power available to the database on the same node. Or, other nodes might be interfering with the network capacities the cluster is using.

During the tests there was a clear indication that at some points, a background process was impacting performance. This process is still not identified. It can be detected by running the same simulation twice in a row and comparing the results. A significant difference can be seen. This difference fades out with time and thus gets smaller and smaller. When this was detected, care was taken not to run any simulations at that moment. However, care should also be taken when interpreting the results. There could always be a result during which there was interference from some process.

Ideally, the simulations would be repeated multiple times with a clean database. However, this is very time consuming and was not possible during the time span of the project. Because of this, significant efforts have been made to keep checking if the results received make sense and if the simulation should be repeated at a later time to verify.

7 Conclusion

During this project two key-value databases were investigated: RAMCloud and Riak. As RAMCloud was written in C++, it's integration went much more smoothly than the Riak integration for which no future proof C/C++ client could be found. Being able to integrate Riak as a Java client did show the flexibility of the ALFA framework.

A lot of simulations were run using different setups of Riak, of the Java clients and of the configurations. These data now show the trend lines and sometimes exact information about certain uses cases of a Riak database. However, more testing should be done using the exact access pattern and value distribution needed. For this to work, a test should be done that measures this exact data to provide as input to the performance framework. As this performance framework has been designed to be easily configurable and extendable, this should be a straightforward step.

In this project, we investigated the difference between read and write performance, the optimal simulation size to get valid data, the usage of different storage backends, the influence of different object sizes, the influence of different cluster sizes, the performance difference with using more relaxed consistency settings, the impact of using security features, the bottlenecks in the designed software and the availability and fault tolerance of Riak. All these results were carefully balanced taking into account several difficulties of testing key-value databases.

Some future work is still possible such as testing other databases, making a C/C++ integration for Riak, deeper investigations into security, using different and more advanced load balancer mechanisms and most importantly testing according to the actual specific access pattern and value distribution that will be used.

This document can now serve as a guideline indicating several important key-value database performance implications, implementation specification for a Java interface to Riak and an implementation specification for the developed performance framework.

8 Bibliography

[1] ALICE, “ALICE Technical Design Report”, 2015

[2] R. Grosso, “Database and data flow for conditions in Run 3”, 2014

[3] RAMCloud, “RAMCloud project page”, https://ramcloud.atlassian.net/wiki/display/RAM/RAMCloud

[4] “The RAMCloud Storage System,” *ACM Transactions on Computer Systems*

[5] Basho, “Basho” http://basho.com/

[6] Basho, “Riak Documentation” http://docs.basho.com/riak/latest/

[7] Basho-labs, “riak-c-client” https://github.com/basho-labs/riak-c-client

[8] Algernon, “riack” https://github.com/algernon/riack

[9] ajtack, “riak-cpp” https://github.com/ajtack/riak-cpp

[10] Basho, “riak-java-client” Available: https://github.com/basho/riak-java-client

[11] Tom Van Steenkiste, “RiakJavaC” https://github.com/tdvsteen/RiakJavaC

[12] M. Al-Turany, “ALFA: Next generation concurrent framework for ALICE and FAIR experiments”

[13] ZeroMQ, “ZeroMQ Guide Chapter 3” http://zguide.zeromq.org/php:chapter3

[14] Tom Van Steenkiste, “KeyValueClusterPerf” https://github.com/tdvsteen/KeyValueClusterPerf

[15] J. Sheehy, D. Smith, “Bitcask, A log-structured hash table for fast key/value data,” *Basho Technologies*

[16] ZeroMQ, “ZeroMQ” http://zeromq.wdfiles.com/local--files/sandbox%3Adealer/fig1.png

Appendix A – Using RiakJavaClient

To execute RiakJavaClient, the correct configuration in the project should first be supplied. Currently, it is configured by in-code parameters. However, this can very easily be updated to use an external format of configuration file according to standard of the system in which it will be used.

In the main function, the list of addresses on which the Riak cluster can be contacted should be supplied. If the nodes in the Riak cluster are changed, this list should also be updated. In the run function of the MessageListener, a connection to the load balancer should be supplied. This consists of the address and a port on which it is accessible. Again, this is very easily updated to use external configuration files.

Before executing the RiakJavaClient, the load balancer should first be deployed. To execute the load balancer from the command line, execute the command in Listing 4. Make sure the hwbroker.jar and zmq.jar files are in the directory and that the ZMQ and JZMQ libraries are installed and accessible. If SHA1 checking should be simulated, append "secure" to the command in Listing 4 and if debug information should be visible, append "debug". Be careful using the debug command as it will print out information each time it receives a message.

```
java -cp zmq.jar:hwbroker.jar hwbroker.HwBroker
```

Listing 4 - Starting the load balancer

After starting up the load balancer, you can start up one or multiple instances of RiakJavaClient. To start an instance, use the command in Listing 5. Make sure the generated jar of the RiakJavaClient project and the generated libraries folder is present in the directory and that the ZMQ, JZMQ and Google Protocol Buffers libraries are installed and accessible. If you want to simulate writes or reads with no consistency verification settings in Riak, append "writeQuorumOff" or "readQuorumOff" to the command in Listing 5. If more advanced consistency settings are needed, the application can easily be updated to accept input parameters from the command line to specify these settings.

```
java -cp ./lib/*:RiakJavaClient-1.0-SNAPSHOT.jar com.cern.riakjavaclient.main
```

Listing 5 - Starting RiakJavaClient

Appendix B – Using KeyValueClusterPerf

To run a KeyValueClusterPerf simulation, one or more instances of the application are required. They can be executed manually but there are also a few bash scripts available to automate the simulation. First the application has to be build. This can easily be done by executing the "make" command.

Next, the hosts on the worker nodes have to be started. This can be done by executing the command in Listing 6 with as PORTNUM and DPORTNUM the communication port and data port respectively to be used. These are ports the controller instance will connect to, so you should make sure that the firewall allows communication across these ports. When multiple worker

instances are used, only one of them has to do the initialisation of the key value database as explained in section 4.1 so the initialisation can be skipped on the other nodes by adding the "--skipinitialisation" flag.

```
./keyvalueclusterperf --port PORTNUM --dport DPORTNUM
```

Listing 6 - Starting a worker instance of keyvalueclusterperf

The process described in the previous paragraph can easily be automated to execute multiple worker instances on the same node by running the "multiRun.sh" script which can be found in the keyvalueclusterperf repository. To run this script, specify the lowest port number for commands and the lowest port number for data in the script itself and supply the amount of instances to launch as a command line argument. This argument to specify the amount of hosts is useful when running simulations.

The final step to run the simulation is to launch the controller instance. This instance will then start contacting all worker nodes to perform the simulation. To know which nodes to contact, a host file has to be specified. This file contains entries which specify the nodes on which instances are running as well as the ports on which they are running. First, the hostname is specified. Then, the first port number is specified followed by the last port number. The last specification is the first data port number. As the previous two port numbers already showed the number of active instances, the last data port number is not specified. An example file can be seen in Listing 7. This host file specifies that on the localhost, 5 instances are running with command ports 2000 to 2004 and data ports 3000 to 3004.

```
#This is a comment line
#hostname:firstCommandPort|lastCommandPort|firstDataPort
localhost:2000|2004|3000
```

Listing 7 - Example host file

Finally, the simulations themselves can also be automated. A script called "multisim.sh" is provided in the keyvalueclusterperf repository to automatically test a key value database in a certain configuration with varying number of simultaneous hosts and varying value sizes. Note that when executing this script, a password for the private key will be asked to be able to connect to the nodes running the keyvalueclusterperf instances. The script uses several Ansible playbooks. The first step is to update all nodes to the correct value size distribution using an Ansible playbook to push an updated file. The following step is to start the keyvalueclusterperf instances on the remote nodes by using the cmd-startkeyvalueclusterperf.yml Ansible playbook. When this is finished, the controller instance is activated locally, or on a remote node. At the end of the simulation, all remaining keyvalueclusterperf instances on the nodes are stopped and the next simulation is started.

Apart from playbooks to start and stop the keyvalueclusterperf instances, there are also playbooks to deploy new builds of the software or new configuration files. These scripts can be found on the GitHub repositories of the respective projects in the scripts-folder.