HOBBIT

Collaborative Project

Holistic Benchmarking of Big Linked Data

Project Number: 688227 **Start Date of Project:** 2015/12/01 **Duration:** 36 months

Deliverable 2.1
Detailed Architecture of the HOBBIT Platform

Dissemination Level	Public
Due Date of Deliverable	Month 9, 31/08/2016
Actual Submission Date	Month 9, 11/08/2016
Work Package	WP2 - Benchmarking Platform
Task	T2.1
Type	Report
Approval Status	Final
Version	1.0
Number of Pages	31

Abstract: This deliverable presents the architecture of the HOBBIT platform.

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 688227

History

Version	Date	Reason	Revised by
0.1	24/05/2016	First draft created	Michael Röder
0.2	30/06/2016	Alpha version created	Michael Röder
0.9	01/08/2016	Beta version created	Michael Röder
0.10	03/08/2016	Reviewed version created	Martin Strohbach
1.0	03/08/2016	Final version created	Michael Röder

Author List

Organization	Name	Contact Information
InfAI	Michael Röder	roeder@informatik.uni-leipzig.de
InfAI	Axel-Cyrille Ngonga Ngomo	ngonga@informatik.uni-leipzig.de
AGT	Martin Strohbach	MStrohbach@agtinternational.com

Executive Summary

This document describes the architecture of the HOBBIT platform in detail. It gives a brief introduction before it describes the main use cases of the platform. In Section 3, an overview is given before the single components are explained in Section 4. The technologies used for the platform are discussed in Section 5. Subsequently, the deployment strategy of the online instance is described in 6 before security aspects are discussed in Section 7.

Contents

1 Introduction

This document describes the architecture of the HOBBIT platform. The platform serves as a framework for benchmarking Linked Data systems. Both, benchmarks focusing the evaluation of the quality of a system using single consecutive requests can be run on the platform as well as benchmarks aiming at the efficiency, e.g., by generating a lot of parallel requests leading to a high work load. Especially for the latter case, the platform should support the handling of Big Linked Data to make sure that even for performant systems a maximum load can be generated.

The HOBBIT platform included in the HOBBIT project aims at two goals. Firstly, we want to offer an open source evaluation platform that can be downloaded and executed locally. Secondly, we want to offer an online instance of the platform for a) running public challenges and b) making sure that even people without the required infrastructure are able to run the benchmarks they are interested in.

The platform comprises of several components. The single components will be implemented as independent containers. The communication between these components is done via a message bus. We will use Docker as a framework for the containerization and RabbitMQ as message bus which is discussed in 5.1 and 5.2, respectively.

This document is structured in the following way. The use cases for the platform are described in Section 2. In Section 3, an overview is given before the single components are explained in Section 4. The technologies used for the platform are discussed in Section 5. Subsequently, the deployment strategy of the online instance is described in Section 6 before security aspects are discussed in Section 7.

Throughout the document, the term "system" refers to the system that is benchmarked and an experiment is a single execution of a benchmark to evaluate a single system.

2 Use Cases

The system as well as the benchmark are modeled as actors with which the platform interacts.

2.1 Benchmark a System

1. The user chooses a benchmark.
    - The platform loads the parameters of the benchmark, e.g., how much data will be used for the benchmark.
    - The platform loads a list of available systems that fit to this benchmark.
2. The user configures the benchmark, chooses a system and starts the benchmark. (If the user is the organizer of a challenge, he might add the date when the challenge benchmark should be executed.)
    - The platform creates the system.
    - The platform instantiates the benchmark controller using the parameters for the execution of the benchmark.
3. The benchmark has finished its work.

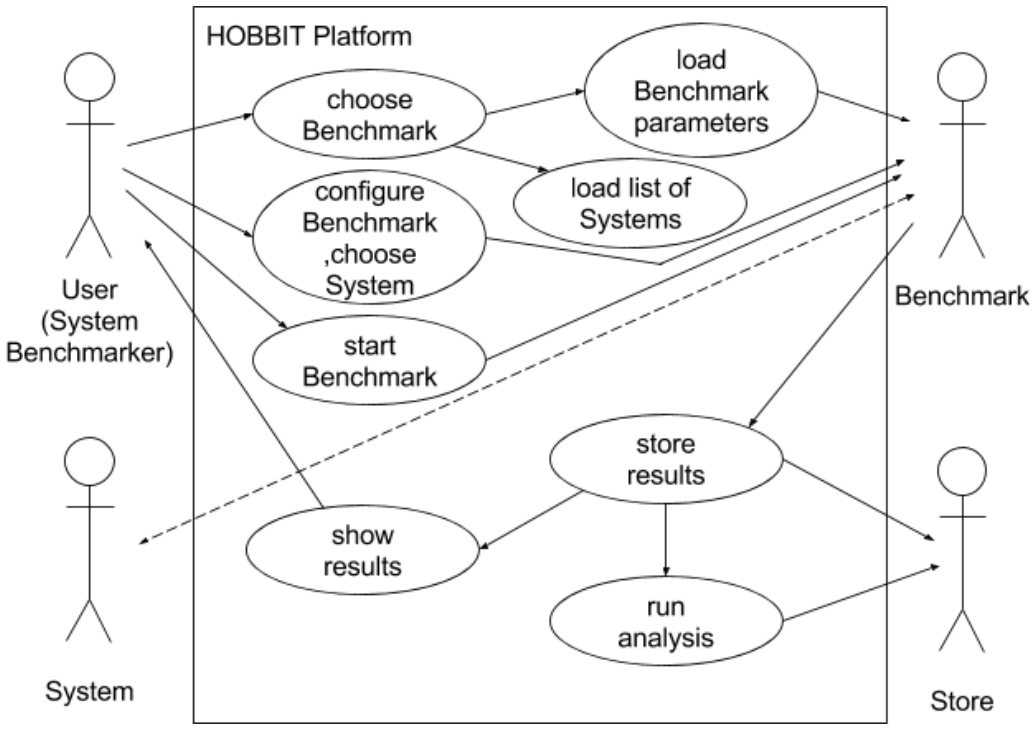

Figure 1: Use case 2.1.

- The platform stores the results.
- The platform starts the analysis module using the results.
- The platform presents the results to the user.

2.2 Show and Compare Benchmark Results

The platform is developed to enable a user to see the performance of a system or to compare several systems with each other.

1. The user chooses a benchmark (and a version of the benchmark).
   - The platform loads those systems (and their versions) for which results are available.
2. The user chooses one or more systems (and their versions).
   - The platform loads the results from the store and shows them.
3. The user wants to create a citable URL for the comparison he created.
   - The platform checks whether there already is a URL for this view inside the store.
   - If such a URL does not exist, the platform creates a new URL and stores it.
   - The platform shows the URL to the user.

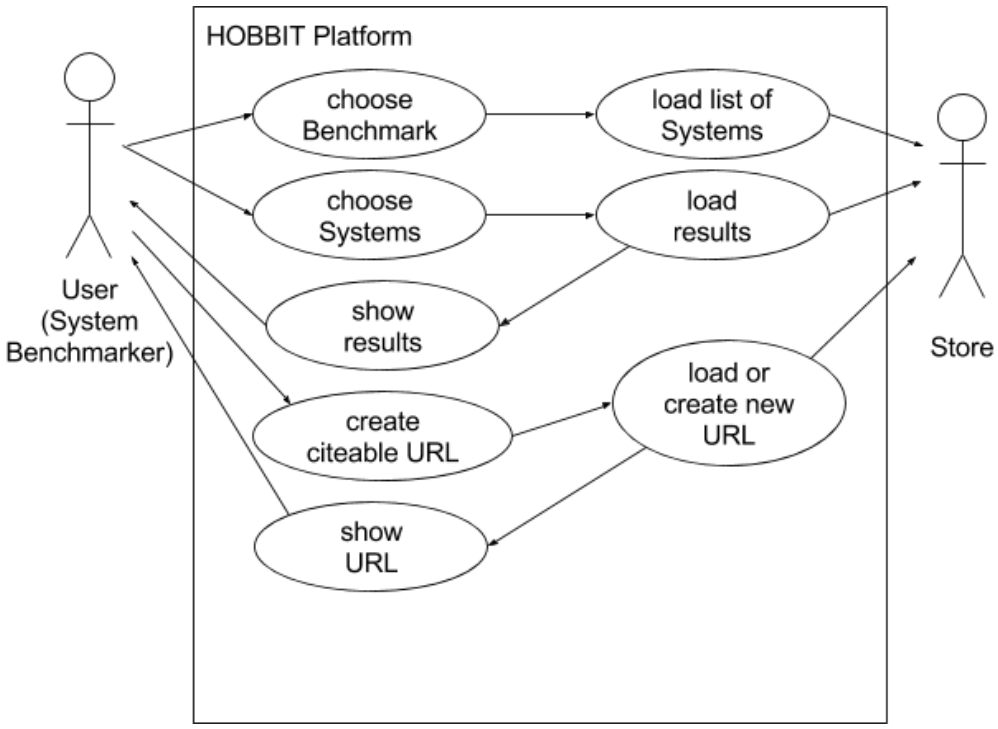

Figure 2: Use case 2.2.

2.3 Add a System

A system provider wants to benchmark his system using the platform and thus needs to upload his system to the platform. Therefore, he has to provide a docker image of his system.

1. Encapsulate the system into a docker container.
2. Write a system adapter that serves as a proxy between the benchmark and the system.
3. Use an interface (either command line or a graphical user interface) to upload the images.

The HOBBIT project foresees to create a) a detailed documentation of these steps and b) a tutorial that will explain the guidelines based on examples.

3 Overview

The HOBBIT platform can be separated into two parts. The first part comprises of platform components that are always running. The second part contains all components that belong to a certain experiment, i.e., the bechmark components as well as the benchmarked system. Picture 3 gives an overview over the components and their relations. Platform components are marked blue while the orange components are the components of a certain benchmark. The benchmarked system does neither belong to the group of benchmark components nor to the platform components.

The data flow shown in the overview picture is carried out using the message bus. However, the HOBBIT platform is modular in a way that allows the benchmark to use a different middleware. Thus, a benchmark might use network sockets or a streaming framework for the communication between the

benchmark components and the communication with the system. For the communication with the platform controller component it has to use the offered RabbitMQ queues. However, throughout the document it is assumed that the benchmark components rely on RabbitMQ as well.

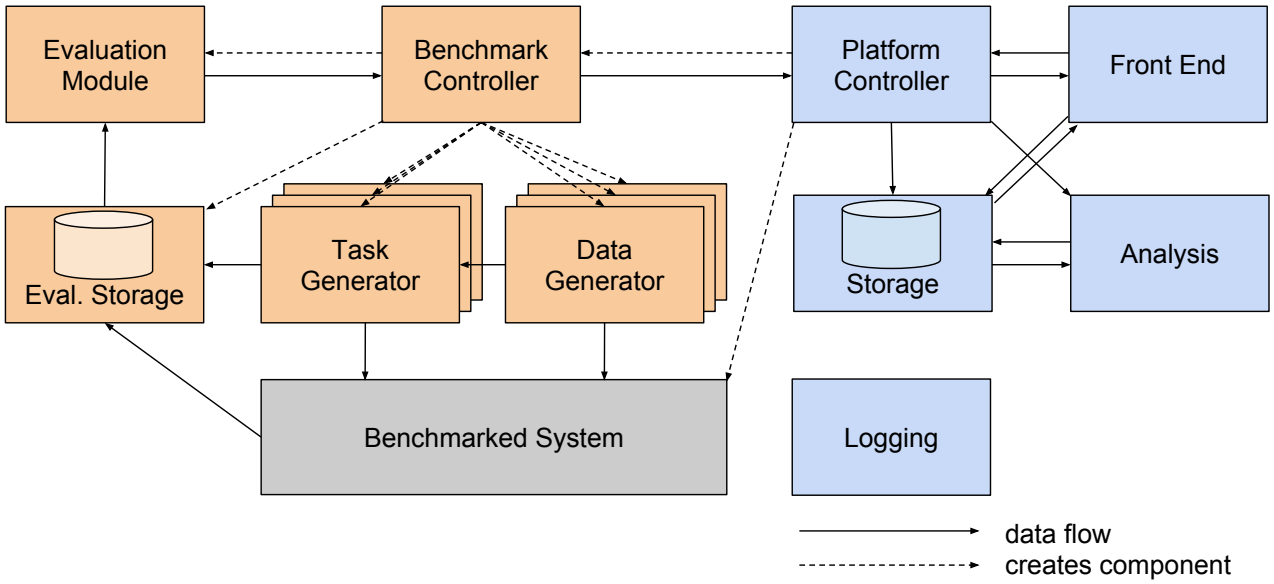

Figure 3: Overview of the platform components.

3.1 Benchmarking a System

In this section the general workflow of benchmarking a system is described. For keeping the workflow simple, the creation of Docker containers is described as being carried out directly by a component while it is discussed in section 7.2 that it will be implemented to be done via the platform controller component. Figure 4 shows a sequence diagram containing the steps as well as the type of communication that is used. However, the orchestration of the single benchmark components is part of the benchmark and might be different.

1. The platform controller makes sure that a benchmark can be started. This includes a check to make sure that all nodes of the cluster are available.
2. The platform controller generates the system that should be benchmarked.
   - The system initializes itself and makes sure that it is working properly.
   - It sends a message to the platform controller to indicate that it is ready.
3. The platform controller generates the benchmark controller.
   - The benchmark controller generates the data and task generators as well as the evaluation storage.
   - It sends a message to the platform controller to indicate that it is ready.
4. The platform controller waits until the system as well as the benchmark controller are ready.
5. The platform controller sends a start signal to the benchmark controller which starts the data generators.

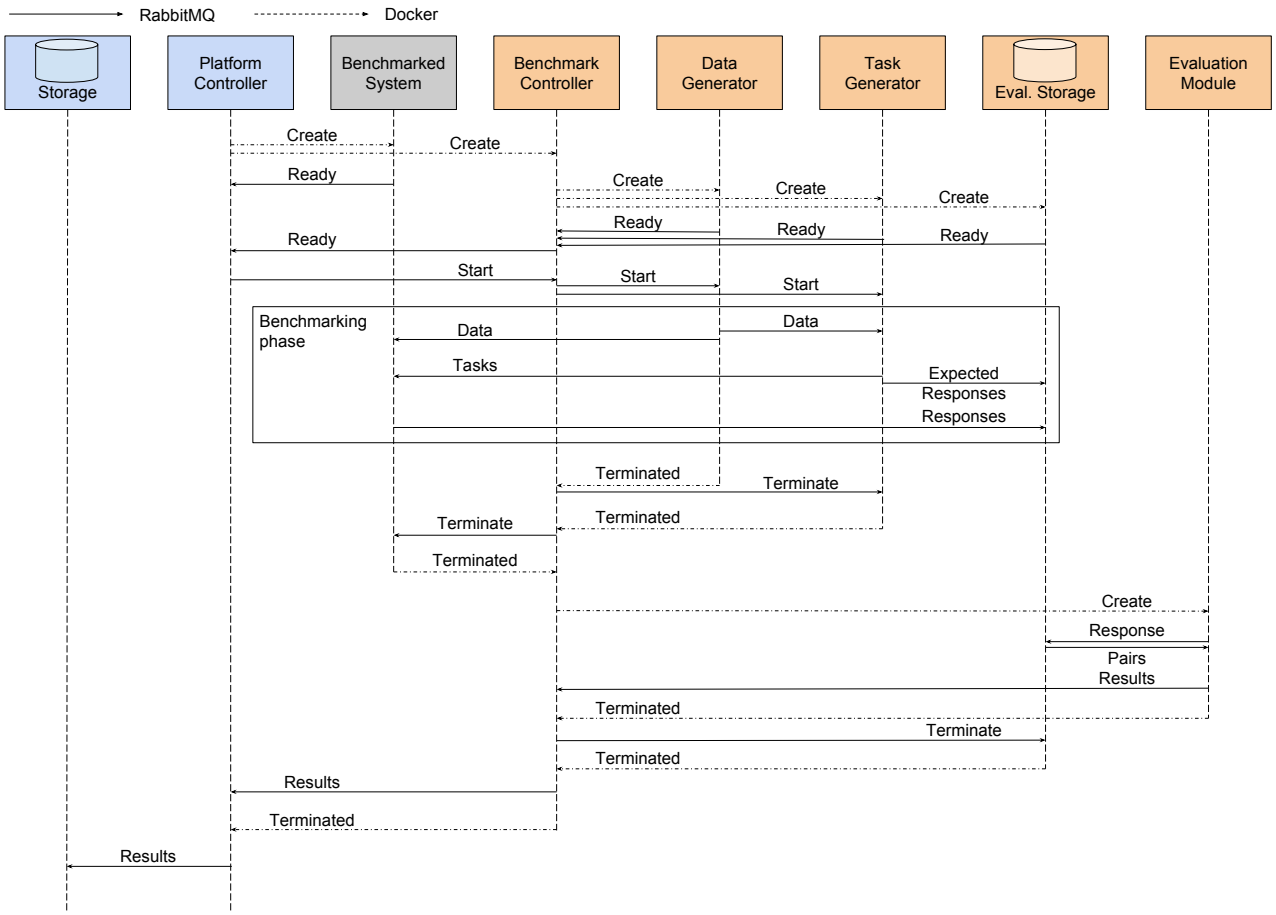

Figure 4: Overview of the general benchmarking workflow. For keeping simplicity, the benchmarked system as well as the front end are left out.

6. The data generators start their mimicking algorithms to create the data.
   - The data is sent to the system and to the task generators.
   - The task generators generate the tasks and send it to the system.
   - The systems response is sent to the evaluation storage.
7. The task generators store the expected result in the evaluation storage.
8. After the data and task generators finished their work the benchmarking phase ends and the generators as well as the benchmarked system terminate.
9. After that the terminated components are discarded and the benchmark controller creates the evaluation module.
10. The evaluation module loads the results from the evaluation storage. This is done by requesting the results pairs, i.e., the expected result and the result received from the system for a single task, from the storage. The evaluation module uses these pairs to evaluate the systems performance and calculate the Key Performance Indicators (KPIs). The results of this evaluation are returned to the benchmark controller before the evaluation module terminates.
11. The benchmark controller sends the signal to the evaluation storage to terminate.

12. The benchmark controller sends the evaluation results to the platform controller and terminates.
13. After the benchmark controller has finished its work, the platform controller can add additional information to the result, e.g., the configuration of the hardware, and store the result. After that, a new evaluation could be started.
14. The platform controller sends the URI of the experiment results to the analysis component.
15. The analysis component reads the evaluation results from the storage, processes them and stores additional information in the storage.

Beside the described orchestration scheme the platform will support other schemes as well. For example, it will be possible to generate all the data in a first step before the task generators start to generate their tasks based on the complete data. In another variant, the task generators are not only sending the generated task but are waiting for the response before sending the next task.

4 Components

In this section the single components are described.

4.1 Platform Components

The platform has several components that are started and stopped together with the complete platform.

4.1.1 Platform Controller

The platform controller is the central component of the HOBBIT platform coordinating the interaction of other components if needed. This mainly includes the handling of requests that come from the front end component, the starting and stopping of benchmarks, the observing of the cluster health status and the triggering of the analysis component.

4.1.1.1 Persistent Status

The internal status of the platform controller will be stored in a database. This enables the shut down or restart of a platform controller without loosing the current status, e.g., benchmarks that have been configured and scheduled inside the queue.

4.1.1.2 Queue

The platform controller manages a queue that contains user configured experiments that will be run in the future. The execution order of experiment configurations is determined using the following features:

- The resources needed for an experiment. If an experiment only needs half of the available cluster nodes, the remaining nodes might be used for a different experiment that fits into the remaining resources.

- The time at which they have been configured by the user (first in first out principle).
- Whether the experiment is part of a scheduled challenge.

The platform controller should be able to guarantee, that the experiments of a certain challenge are executed at a certain date. Thus, it might be required to check the queue for experiment configurations that are not at the head of the queue, but need to be executed because of this constraint.

Additionally, it should be possible for an administrator to block the execution of new experiment configurations, e.g., in cases in which the system should be shut down or restarted.

4.1.1.3 Interaction with Front End

The platform controller interacts with the front end in the following situations.

1. The front end shows a configuration screen to the user with which the user can start a benchmark with a particular system. The platform controller has to give the following information to the front end:
   - Available benchmarks
   - Available systems (after the benchmark has been chosen)
   - Benchmark parameters (and their default values) (after the benchmark has been chosen)
2. The front end sends a configuration for an experiment comprising
   - the name of a benchmark,
   - its parameter values,
   - the name of the system
   - a notification method, e.g., mail address of the user under which the user wants to be informed about his experiment (optional parameter).
3. The platform controller should be able to send the status of its experiment queue to the front end, if the user wants to see the position of its experiment in the queue.

4.1.1.4 Cluster Health Observation

The platform controller has to be able to observe the status of the cluster that is used to execute the experiments. If one of the nodes drops out of the cluster, the comparability between single experiments is not given. Thus, the platform controller needs to be aware of the number of working nodes that are available for the experiment. It uses the Docker Swarm API to check the availability of the single nodes.[1]

4.1.1.5 Benchmark Execution

The platform controller contains a queue of experiments, i.e., benchmark and system combinations. If there is no running experiment and the queue is not empty, the platform controller should initiate the execution of an experiment in the following way:

[1] Since version 1.12 Docker Swarm has embedded health checks and failover policies. https://docs.docker.com/engine/swarm/

1. The platform controller makes sure that a benchmark can be started. This includes a check to make sure that the system/cluster is healthy.
2. The platform controller generates a unique HOBBIT ID for the experiment.
3. The platform controller generates the system.
4. The platform controller generates the benchmark controller.
5. The system as well as the benchmark controller load all the data that they need and make sure that they are working properly. Afterwards they send a message to the platform controller using the command queue to indicate that they are ready. The platform controller is waiting for these messages.
6. After both the system and the benchmark controller are ready, the platform controller sends a start signal to the benchmark controller. With this signal, the platform controller stops observing the system. It is assumed that the benchmark controller takes care of sending a termination signal to the system.
7. The platform controller receives the results from the benchmark controller.
8. The results might be extended with additional information about the hardware before they are send to the storage.
9. A notification might be sent to the user (e.g., via mail).

The platform controller observes the state of the benchmark controller. If the experiment takes more time than a configured maximum, the platform controller terminates the benchmark controller and all the containers that belong to it. Since the platform controller manages the creation of containers it has a list of created containers that belong to the currently running benchmark.

4.1.1.6 Interaction with Analysis Component

The platform controller needs to be able to trigger the analysis component sending a benchmark controller and a system name to inform the analysis component that new results for this combination are available.

4.1.1.7 Docker Container Creation

The platform controller is the only component that has direct access to the docker daemon. If another component would like to start a docker container, it has to send a request to the platform controller containing the image name and parameters. Thus, the platform controller offers the central control of commands that are sent to the docker daemon which increases the security of the system.

4.1.2 Storage

The storage component contains the experiment results. It comprises two containers—a triple store that uses the HOBBIT ontology to describe the results and a Java program that handles the communication between the message bus and the triple store.

4.1.2.1 Read Access

The storage component offers a public SPARQL Endpoint with read-only access. Additionally, the front end needs to be able to load results to present them to the user while the analysis component needs the results for deeper analysis.

4.1.2.2 Write Access

The storage component offers write access to the benchmark controller for storing experiment results. Additionally, the analysis component needs to be able to store additional results of the result analysis. To make sure that the write access is limited to these two components the storage component offers a secured communication using a generated key as described in section 7.3. The generation and exchange of keys is managed by the platform controller component. During the initialization of the storage component, the platform controller generates a key and adds it to the system environment variables of the storage and analysis component.

4.1.2.3 Ontology

The experiment results as well as the results of the analysis component will be stored as RDF triples in the storage component. Therefore, an ontology is needed to define how the data can be formulated in RDF. This ontology might reuse existing or define new vocabularies if needed, e.g., the LDBC Ontology[2].

The following requirements have to be fulfilled by the ontology:

- It must offer classes and properties to store the configuration and the result of an experiment
  - The single benchmarks and systems need URIs for representing them in the triple store.
  - The KPIs have to be stored.
  - The cluster and its configuration with which the experiment has been carried out has to be described.
  - The benchmark configuration needs to be stored.
- The ontology should support the work of the analysis component.
  - It should be possible to define the features of a system and a benchmark.
  - The KPIs should be described with their type.

4.1.3 Front End

The front end component handles the interaction with the user. It offers different functionalities to the different user groups. Thus, it contains a user management that allows different roles for authenticated users as well as a guest role for unauthenticated users.

[2] http://ldbcouncil.org/sites/default/files/LDBC_D1.1.7.pdf

4.1.3.1 Roles

Guest. A guest is an unauthenticated user. This type of user is only allowed to read the results of experiments and analysis.

System provider. This user is allowed to upload system images and start benchmarks. Additionally, it is allowed to combine experiment results and store the created view using a generated, citable URI.

Benchmark provider. This user has the same rights as a system provider but is allowed to upload benchmark images.

Challenge Organizer. This user is allowed to organize a single challenge, i.e., define experiments with a certain date at which they will be executed.

Admin. Admins are allowed to add, edit or remove users as well as challenges. Additionally, they can stop the platform.

4.1.3.2 Docker Images

The front end offers the possibility to upload new system and benchmark images. These images are stored in a local Docker repository.

4.1.4 Analysis

This component is triggered after an experiment has been carried out successfully. Its task is to enhance the benchmark results by combining them with the features of the benchmarked system and the data or task generators. These combination can lead to additional insights, e.g., strengths and weaknesses of a certain system.

While the component uses the results of a benchmark it is modelled independently from any benchmark implementation. Thus, the analysis has to be implemented in a general way. It uses the knowledge gained from the system and benchmark described using the HOBBIT ontology and decides which analysis it could execute. An example of such a behaviour is Weka[3] which offers machine learning algorithms based on the input and expected output data. Since the implementation of this component heavily depends on the HOBBIT ontology and the knowledge that can be expressed by it, these two parts should be developed closely.

4.1.5 Message Bus

This component contains the message bus system. There are several messaging systems available from which we chose RabbitMQ. This is discussed in detail in section 5.2.

Table 1 shows the different queues that are used by the platform components. We will use the following queue types:

- **Simple.** A simple queue has a single sending component and a single receiving consumer.

[3] http://www.cs.waikato.ac.nz/ml/weka/

- **Bus.** Every component connected to this queue receives all messages sent by one of the other connected components.
- **RPC.** The queue has one single receiving consumer that executes a command, e.g., a SPARQL query, and sends a response containing a result.

Queues that are used exclusively by the benchmarking components or the benchmarked system are not listed in table 1, since their usage depends on the benchmark. However, if the benchmark relies on RabbitMQ, the benchmark implementation has to add the ID of the experiment to the queue name to enable a parallel execution of benchmarks.

Name	Type	Description
`hobbit.command`	Bus	Broadcasts commands to all connected components.
`hobbit.storage_read`	RPC	The storage component accepts reading SPARQL queries. The response contains the query result.
`hobbit.storage_write_<key>`	RPC	The storage component accepts reading and writing SPARQL queries. The response contains the query result. The `<key>` is a security feature describe in Section 7.3.
`hobbit.controller`	RPC	Used by the front end to interact with the controller as described in section 4.1.1.3.

Table 1: List of queues used inside the platform.

The `hobbit.command` queue is used to connect the loosely coupled components and orchestrate their activities. Since the platform should be able to support the execution of more than one experiment in parallel, we will use a simple addressing scheme to be able to distinguish between platform components and benchmark components of different experiments. Table 2 shows the structure of a command message. Based on the HOBBIT ID at the beginning of the message, a benchmark component can decide whether the command belongs to its experiment. The message contains a byte that encodes the command that is sent. Based on this command a component that belongs to the addressed experiment decides whether it has to react to this message. Additional data can be appended as well if necessary.

Bytes	Type	Description
0..3	int	Length h of the HOBBIT ID.
4..$h+3$	String	HOBBIT ID of the experiment this command belongs to or "SYS" for a message between two platform components.
$h+4$	byte	ID of the command.
$> h+4$	byte[]	Additional data (optional)

Table 2: Structure of a message of the `hobbit.command`.

4.1.6 Logging

The Logging comprises of three single components—Logstash, Elasticsearch and Kibana. While Logstash is used to collect the log message from the single components, Elasticsearch is used to store them inside a fulltext index. Kibana offers the front end for accessing this index.

4.1.6.1 Logging inside Components

The single components should write log messages to the standard output. The docker containers will be configured in a way that sends these standard outputs to the Logstash instance.

We encourage the usage of a facade, e.g., slf4j[4], in the program code. This offers the advantage that a certain implementation, e.g., log4j, can be chosen without influence on the already written code. Additionally, all components should share a single pattern for the log messages to ease the analysis of log messages and the configuration of Kibana.

4.2 Benchmark Components

These components are part of the benchmark. They are instantiated for a particular benchmark and should be destroyed when the benchmark terminates.

4.2.1 Benchmark Controller

The benchmark controller is the central component of an evaluation. It creates and controls the data generators, task generators, evaluation-storage and evaluation-module.

4.2.1.1 Workflow

1. Is created by the platform controller with the following parameters
   - HOBBIT ID of the experiment
   - benchmark specific parameters
   - the Docker container ID of the system
2. The benchmark controller initializes itself and connects to the `hobbit.command` queue.
3. The benchmark controller waits for the system to send the start signal (`hobbit.command`).
4. It creates the data and task generators as well as the evaluation storage (Docker). It waits for the single components to report that they are ready (`hobbit.command`).
5. It sends a start signal to the created benchmark components (`hobbit.command`).
6. It waits for all data generators to end (Docker).
7. It sends a signal to the task generators that all data generators have terminated (`hobbit.command`).

[4] http://www.slf4j.org/

8. It waits for all task generators to end (Docker).
9. It sends a signal to the system that all task generators have terminated (`hobbit.command`).
10. It waits for the system to end (Docker).
11. It creates the evaluation module (Docker).
12. It waits for the evaluation module to finish (Docker).
13. it sends a terminate signal to the evaluation module (`hobbit.command`) and waits for it to finish (Docker).
14. It sends the results received from the evaluation module to the platform controller and exits with code 0.

4.2.1.2 Error Handling

The error handling inside the benchmark components is important since the occurrence of an error can lead to a wrong evaluation result. Thus, the benchmark components need to have exact rules how they should act in case of a severe error.

All benchmark components are observed by another component that waits for their termination. Thus, if a benchmark component encounters a severe error that can not be handled inside the component and will lead to a wrong benchmark result, the component should terminate with an exit code > 0.

The benchmark controller will be informed by the platform controller that one of its component exited with an error code. The benchmark controller has to send a stop signal to all benchmark components as well as the system and wait for their termination. If the benchmark controller has been initialized and already sent the message to the platform controller that it is ready, it has to stop the system as well. Before it terminates the benchmark controller should send the error to the platform controller and terminate with a status > 0. The platform controller should store the error in the storage to give the feedback to the user that an error has occurred.

If the benchmark controller has a faulty implementation and does not handle the error in the way described above the platform controller implements a fall back strategy. Firstly, a benchmark that does not terminate during a preconfigured time frame is stopped by the platform controller. Thus, even if the combination of a crashed benchmark component and the faulty error handling lead to a deadlock of the benchmark, the platform itself will continue as soon as the preconfigured run time has been reached. Secondly, the platform controller comprises a mapping of the benchmark components to the HOBBIT ID of the experiment. Thus, even if the benchmark controller is not able to terminate all its components, the platform controller will be able to stop the components using this mapping and the Docker daemon.

4.2.2 Data Generator

The data generator contains a mimicking algorithm implementation that is able to generate the data needed for the evaluation. For benchmarking a system based on big data, the data generator might be instantiated several times. Each instance will receive an ID that should be used to generate different data.

Additionally, the data generation needs to be repeatable, i.e., a data generator that is run a second time with the same configuration has to produce exactly the same data as it did during its first execution. Thus, if the generation is relying on pseudo-random processes a seed should be used.

During a benchmark experiment, the data generator typically performs the following steps

1. It is created by the benchmark controller with the following parameters
   - HOBBIT ID of the experiment
   - ID of this particular generator instance as well as the number of generators
   - benchmark specific parameters (optional)
2. It initializes itself and connects to the `hobbit.command` queue.
3. It sends the ready signal to the benchmark controller (`hobbit.command`).
4. It waits for the start signal (`hobbit.command`).
5. It generates data based on the given parameters.
6. It terminates with status code 0.

Data generator components will be linked to a volume which can be used to cache the generated data. Thus, if the same data generator is executed with the same parameters, the cached data might be reused.

4.2.3 Task Generator

The task generator gets the data from the data generator, generates tasks that can be identified with an ID. This ID is needed to map the system responses to the expected responses during the evaluation, i.e., a task comprising a SPARQL query should have the same task ID as the expected result of the query. The generated tasks are sent to the system while the expected response is sent to the evaluation storage.

During a benchmark experiment, the task generator typically performs the following steps

1. It is created by the benchmark controller with the following parameters
   - HOBBIT ID of the experiment
   - ID of this particular generator instance as well as the number of generators
   - benchmark specific parameters (optional)
2. It initializes itself and connects to the `hobbit.command` queue.
3. It sends the ready signal to the benchmark controller (`hobbit.command`).
4. It waits for the start signal (`hobbit.command`).
5. It generates tasks.
   - It reads incoming data from the data generators.
   - It generates a task and the expected solution.

- It sends the task to the system.
- It sends the expected solution to the evaluation storage.

6. If the `DATA_GEN_TERMINATE` signal is received (`hobbit.command`) it consumes all data that is still available.
7. It terminates with status code 0.

Task generator components will be linked to a volume which can be used to cache the generated tasks. Thus, if the same task generator is executed with the same parameters, the cached tasks might be reused.

4.2.4 Evaluation Storage

The evaluation storage is a component that stores the gold standard results as well as the responses of the benchmarked system during the computation phase. During the evaluation phase it sends this data to the evaluation module. Internally, the component is based on a key-value store and a small Java program that handles the communication with other components. The ID of the task is used as the key, while the value comprises the expected result as well as the result calculated by the benchmarked system.

During a benchmark experiment, the task generator typically performs the following steps

1. It is created by the benchmark controller with the HOBBIT ID of the experiment.
2. It initializes itself and connects to the `hobbit.command` queue.
3. It sends the ready signal to the benchmark controller (`hobbit.command`).
4. It reacts to all incoming queues and stores (expected) results. It might add a timestamp to the results received from the system to enable time measurements.
5. After the evaluation module has been started, the evaluation storage will receive a request to iterate the result pairs. Every request will be answered with the next result pair.
6. If a request from the evaluation module is received but can not be answered because all result pairs have been sent, an empty response is sent.
7. If the signal to terminate is received from the benchmark controller it terminates with status code 0.

4.2.5 Evaluation Module

The evaluation module evaluates the result generated by the benchmarked system. Depending on the goals of the benchmark this might be accomplished by using a comparison with the expected results or based on time measurements.

1. It is created by the benchmark controller with the HOBBIT ID of the experiment.
2. It initializes itself and connects to the `hobbit.command` queue.

3. It requests result pairs from the evaluation storage and evaluates them.
4. After the last pair has been received and evaluated, the evaluation results are summarized and sent to the benchmark controller.
5. It terminates with status code 0.

4.3 Benchmarked System Components

The HOBBIT platform does not have any requirements, how a system is structured internally. However, a system that should be benchmarked using the HOBBIT platform has to implement a certain API. Depending on the implementation of the system and the benchmark that will be used, a system could implement the system API directly or use a system adapter that might be executed in an additional Docker container.

The API a system has to implement can be separated into two parts. One part is the communication with the HOBBIT platform while the other part is the communication with the benchmark.

The communication with the HOBBIT platform comprises the following three aspects.

1. With uploading the system, the user will be asked for meta data about the system, e.g., the systems name. This data might contain parameters of the system that could be used by the analysis component.
2. During the creation of the systems Docker container the platform controller sets environment variables which have to be used by the system, e.g., the ID of the current experiment, the system is taken part of.
3. The system has to connect to the `hobbit.command` queue and send a signal that indicates that it is ready of being benchmarked.

The second part of the API relies on the benchmark. As described in Section 3 the data generators might send data while the task generators might send tasks. The responses generated by the system should be send to the evaluation storage. This communication might be carried out using RabbitMQ. However, as described before, the benchmark implementation might use a different communication which has to be used by the system as well.

5 Technologies

In this section, the single technologies that are used for the HOBBIT platform are discussed.

5.1 Container Framework

The use of containerization frameworks for HOBBIT platform instead of traditional virtual machines (VM) was made because containers can enable the platform to run with a significantly smaller overhead. VMs run a virtual copy of all the allocated hardware, capture the allocated system resources completely and run a full copy of an operating system. In contrast, containerization frameworks utilize the host operating system, and make use of shared supporting programs and libraries as well as system resources.

The current containerization landscape is represented by several container systems, starting from the most low level ones like LXC[5] or runC[6] and to the high level solutions like Docker[7] or Rkt[8]. Using low level frameworks would mean developing infrastructure and supporting services (e.g. networking between several machines) on our own. Thus we are not considering this option and will only apply higher level solutions.

Even though there are more high level solutions than Docker and Rkt, almost all of them are still in their infancy and have not reached first stable release versions. Since the container framework is going to be a critical part of the HOBBIT architecture, we have only considered stable and tested frameworks—Docker and Rkt. At the time of reviewing the frameworks for the project, Docker has been at version 1.9 while Rkt has only released its first release candidate. Even though Rkt has some interesting features and ideas, it has not been tested as extensively as Docker purely due to the number of large companies using the latter every day. Thus we decided in using Docker, which has all the high level features we require for the HOBBIT platform and which is a stable and tested framework used and trusted by many large enterprise companies.

5.2 Message Bus

There is a variety of messaging brokers that can be used as an internal message bus within the HOBBIT project. Requirements include message persistence, durability, smart routing and support for several exchanges and queues. The most popular solutions that support those features are RabbitMQ[9] and Apache Kafka[10].

RabbitMQ is one of the leading open-source messaging systems. It is written in Erlang, implements the Advanced Message Queuing Protocol (AMQP) and is one of the most used messaging brokers in large enterprise companies. It supports both message persistence and replication, and is very well documented. It uses message acknowledgements to ensure delivery state on the broker itself. RabbitMQ provides a wide variety of routing strategies—from direct exchanges to topic routing. It is also possible to cluster several RabbitMQ servers together, forming a single logical broker that can handle a larger volume of messages.

Kafka is a newcomer in the area of messaging brokers. It takes a different approach to messaging—the server itself is a streaming publish-subscribe system that utilizes Apache Zookeeper[11] for distributed coordination. The only available routing mechanism is topic routing, meaning that Kafka is not designed for granular routing. Additionally, messages aren't acknowledged on a server, but instead message offsets processed by consumers are written to Zookeeper, either automatically in the background, or manually, which allows Kafka to achieve very high performance. Since Kafka provides message ordering inside partitions, the Kafka consumers have to be smart enough and resolve ordering themselves which brings additional burden to the development process of clients. It is also worth noting that Kafka does not have a stable release version (v1.0) yet.

For the HOBBIT project we have picked RabbitMQ primarily because of its maturity and flexible routing possibilities. While Apache Kafka might have been easier to scale, potential issues with routing and an increased effort required to develop "smart" clients outweigh that advantage. In addition,

[5] https://linuxcontainers.org/
[6] https://github.com/opencontainers/runc
[7] https://docker.com/
[8] https://coreos.com/rkt/
[9] https://www.rabbitmq.com/
[10] http://kafka.apache.org/
[11] https://zookeeper.apache.org/

RabbitMQ has a larger community and better documented edge cases.

5.3 Triple Store

Table 3 lists the state-of-the art triple stores. In order to choose the appropriate solution the following features were evaluated:

- **Name**: The name of the triple store.
- **Storage Type**: Refers to the technical information about the way each system uses to store its data. For example, whether data is represented in grah structure with nodes and edges (Graph DBMS) or in large relational tables (RDF Store) etc.
- **Java Client**: Refers to the ability of systems to offer Java APIs in order to be able for the HOBBIT platform to communicate with the triplestore.
- **Transaction concepts**: A crucial requirement for selecting the triplestore is the Backup & Restore mechanism support, so the *Durability* property of *ACID* is utmost importance when evaluating the triplestores.
- **Licensing**: This column indicates the license agreement under which the software is published.
- **Comments**: Important comments that have to be taken into account in order to chose the appropriate solution that fits our requirements.

For the storage component a triple store is needed that 1) can handle millions of triples and 2) is easy to backup. Virtuoso is the system that fits our requirements best, so it is the one that will be used as triple store for the storage component.

Name	Storage Type	Java Client	Transaction concepts	Licensing	Comments
Virtuoso	Hybrid (Relational Tables and Relational Property Graphs)	yes	ACID	GPL v2 or Commercial	/
GraphDB	Triplestore/Quadstore	yes	ACID	Limited Free or Commercial	No more than two queries in parallel can handled in free edition
AllegroGraph	Graph DBMS	yes	ACID	Limited Free or Commercial	Loaded data limited to 5M and 50M on Free and Developer editions
BlazeGraph	Graph DBMS	yes	ACID	GPL v2 or Commercial	/
Stardog	Graph DBMS	yes	ACID	Limited Free or Commercial	No more than 8 concurrent connections, 10 databases and 25M triples/db in free edition
Fuseki/Jena TDB	Triple store	yes	ACID	Apache License v2.0	/
RDFox	In-memory Triple store	yes	no	Academic license[12]	RDFox does not support transactional updates and its in-memory store restrict us for handling big data

[12] http://www.cs.ox.ac.uk/isg/tools/RDFox/RDFox_Academic_Licence.txt

5.4 Key-Value Store

The evaluation storage described in section 4.2.4 is based on a key-value store. Since the evaluation storage needs to be able to handle large amounts of data, the most important requirement for the underlying key-value store is that it needs to be able to store a large amount of data in a short time. Table 4 lists available key-value store solutions. For each system the table lists the following features:

- **Name**: The name of the storage solution.
- **Storage Type**: What kind of paradigm is being used for storage. We typically distinguish the following types:
  - **Database**: The most generic type of store. It can handle any type of storage paradigm.
  - **Key-value**: A store that focuses on storing keys of a certain type and mapping it to values of a certain type. This is the type we are looking for.
  - **Document**: A subtype of the key-value store, which is optimized for handling semi-structured data. Entries in the store are documents that can have different properties. A document store typically stores additional metadata about document properties.
  - **Column-oriented**: Relational database typically store tuples in row-oriented manner. When data access patterns however allow data to be saved in a column-oriented manner, data can be compressed more efficiently which results in less I/O and more parallelizability.
- **In-memory**: Our key-value store must be able to store a lot of data. This is why we need our key-value store to not require everything to be loaded in-memory. Disk storage size is typically much larger than memory size, so the storage solution must be able to load only parts of the large datasets in-memory.
- **Java client**: The HOBBIT platform will be written in the Java language, this is why we require easy interfacing with the key-value storage solution through a Java client.
- **Sharding**: Because we are working with Big Data, it may not be possible for all data to be stored on a single machine. This column indicates if the storage solution allows data to be spread over different machines.
- **Licensing**: We aim to use open software for the HOBBIT platform. This column indicates through license the storage software is published.
- **Comments**: Important comments about the storage solution to take into account.

As it can be seen from the table, Riak fits our requirements best and will be used for the evaluation storage.

Name	Storage Type	In-Mem.	Java Client	Sharding	Licensing	Comments
Riak[13] *(Probably the best option)*	key-value	No	Yes	Yes [1]	Apache 2 (optional commercial support)	Benchmarks are available[14].
Project Voldemort[15] *(Also a viable option, but some versioning overhead)*	key-value	No	Yes	Yes	Apache 2	Might be harder to use in practise than Riak because of the limited documentation and because it is being used less than Riak. Supports versioning, which we don't need, so this will introduce some overhead.
Amazon DynamoDB[16] *(Probably not an option)*	key-value	/	Yes	Yes	Licensing	Hosted and managed by Amazon.
Kyoto/Tokyo Cabinet[17] *(Not an option)*	key-value	No	Yes	No	GNU GPL	Extremely fast

Table 4: Properties of most important technologies that enable key-value storage.

[13] http://docs.basho.com/riak/latest/
[14] https://medium.com/@mustwin/benchmarking-riak-bfee93493419#.z9pximxbn
[15] https://github.com/voldemort/voldemort
[16] http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
[17] http://fallabs.com/kyotocabinet/

Name	Storage Type	In-Mem.	Java Client	Sharding	Licensing	Comments
Redis[18] *(Not an option)*	key-value	Yes	Yes	No, but redis-cluster[19] or twem-proxy[20] could be used.	BSD	Very fast, but requires everything to be loaded in-memory, which is not feasible for big data.
MongoDB[21] *(Not an optimal solution)*	document	No	Yes	Yes	Apache GPL 3	Very efficient *document* store. It is possible to use it as a key-value store, but there are better alternatives. There would be too much overhead of the additional features which we don't need.
LevelDB[22] *(Not an option)*	key-value	No	Yes	No	New BSD	String-based K/V store, no built-in sharding support.
RocksDB[23] *(Not an option)*	key-value	No	Yes	No	BSD	Builds upon LevelDB to improve efficiency when running on systems with many cores.
BigTable[24] *(Probably an option)*	database	No	Yes	Yes	Proprietary, by Google	Hosted by Google

Table 4: Properties of most important technologies that enable key-value storage.

[18] http://redis.io/
[19] http://redis.io/topics/cluster-tutorial
[20] https://github.com/twitter/twemproxy
[21] https://www.mongodb.org/
[22] https://github.com/google/leveldb
[23] http://rocksdb.org/
[24] https://cloud.google.com/bigtable/

Name	Storage Type	In-Mem.	Java Client	Sharding	Licensing	Comments
Couchbase[25] *(Not an optimal solution)*	document	No	Yes	Yes	Free community edition or paid enterprise edition with support	Document-store, so this introduces overhead when compared to pure K/V stores.
Apache Cassandra[26] *(Not an optimal solution)*	column-oriented	No	Yes	Yes	Apache 2	Not a pure K/V store, but column oriented, which we don't need.
MemcacheDB[27] *(Probably not an option, unmaintained)*	key-value	No	Yes	Yes	BSD	Uses Memcached protocol, the project is not being maintained anymore.
HBase[28] *(Not an optimal solution)*	database	No	Yes	Yes	Apache 2	Focuses on big data. Possible overhead because of support for millions of columns, while we only need 2 (3).
CouchDB[29] *(Not an optimal solution)*	document	No	Yes	Yes	Apache 2	JSON-based, can store any type of data, focused on concurrency support.

Table 4: Properties of most important technologies that enable key-value storage.

[25] http://www.couchbase.com/
[26] http://cassandra.apache.org/
[27] http://memcachedb.org/
[28] https://hbase.apache.org/
[29] http://couchdb.apache.org/

Name	Storage Type	In-Mem.	Java Client	Sharding	Licensing	Comments
RethinkDB[30] *(Not an optimal solution)*	document	No	Yes	Yes	Apache 2	Focused on using real-time data to clients, which we don't need.
RebornDB[31] *(Not an option)*	key-value	Yes(?)	Yes, a Redis client	Yes	MIT	Redis protocol, not a lot of information can be found.

Table 4: Properties of most important technologies that enable key-value storage.

[30] https://www.rethinkdb.com/
[31] https://github.com/reborndb

6 Deployment

The online instance of the HOBBIT platform will be deployed on a server cluster. Table 5 contains the different nodes of the cluster, their features and the components that will be deployed on them. However, the distribution of worker nodes, i.e., 4 workers for the system and 2 nodes for benchmark components, might be changed depending on the benchmark.

Node	Features	Components
Data Server	~ 100 TB disk space	– Storage – Evaluation storage – Logging
Master	– 10 cores – 32 GB RAM	– Platform controller – Benchmark controller – Docker – RabbitMQ – Front end – Analysis component
Worker 1 – 4	– 16 hardware cores – Hyperthreading – 256 GB RAM	– Benchmarked system
Worker 5 – 6	– 16 hardware cores – Hyperthreading – 256 GB RAM	– Data generator – Task generator – Evaluation Module

Table 5: Description of cluster nodes and their tasks.

7 Security

In this section, the security of the HOBBIT platform is discussed. One central feature of the platform is the ability to upload systems and benchmarks. However, the execution of third party containers can be risky since an attacker could upload harmful software code. The platform contains several critical components that have to be protected against attacks from outside as well as attacks from inside, i.e., uploaded components.

7.1 Cluster Hardware

A goal of a typical attacker is to gain control of a machine. However, all software components are executed in Docker containers. We will restrict the rights of these containers to exclude the execution

of malicious commands using AppArmor security profiles.[32] Thus, an attacker might have the ability to execute code inside an uploaded container but the process is not able to influence the underlying operating system.

7.2 Docker

An attacker might try to get access to the Docker daemon to download and execute additional containers. This can be prohibited by defining user credentials for the Docker daemon. A container that has to start another container, e.g., the benchmark controller container that has to create the data and task generators, has to send a request to the platform controller. Thus, the platform controller can decide whether a Docker container should be started or not.

7.3 Storages

Another goal of an attacker might be the manipulation of data that is stored either in the central triple store of the platform or in the evaluation storage. Such an attack will be prevented by the following steps.

- Allow the definition of user credentials for the triple store in a non-public properties file making sure that only the Java program of the storage component can write to the triple store.
- In a similar way the evaluation storage component should be able to generate new credentials and apply them to its key-value store every time it is executed.
- Secure the communication between the analysis, platform controller and storage components.
- Secure the communication between the task generator and the evaluation storage to make sure that an uploaded system adapter is not able to insert his own gold standard answers into the evaluation storage.

There are several ways to secure the message bus-based communication between two components. Firstly, the endpoints can secure every single message by using either a symmetric encryption algorithm to encrypt the complete message or an asymmetric encryption to sign the single messages. However, in both cases every message would cause an additional overhead because of encryption and decryption steps.

A second approach is to hide the queue name by adding a random number to the name during its creation. Since a RabbitMQ client is not able to get the list of existing queues, an attacker would have to guess the name of the queue. In this case the random number serves as a key that must be known to a system to connect to the queue.

7.4 User Management

An attacker could try to get access to the user management to gain additional rights. Since we aim to reuse an already established solution for our user management, choosing a secure solution will prevent these attacks.

[32] https://docs.docker.com/engine/security/apparmor/

References

[1] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.