



University of  
Zurich<sup>UZH</sup>

# Investigation of Message Exchange Capabilities of OpenStack

*Simon Balkau*  
*Glarus, Switzerland*  
*Student ID: 09-926-569*

Supervisor: Patrick G. Poullie,  
Date of Submission: June 19, 2017



# Zusammenfassung

Der Fairness Service (FS) ermöglicht Fairness zwischen Cloudbenutzern, in dem durchgehend Informationen über den Ressourcenverbrauch von virtuellen Maschinen gesammelt, die zur einer Gewichtsmetrik der Benutzer kumuliert, die Ressourcen der VMs entsprechend umverteilt, sodass die Gewichtsmetriken der Benutzer angeglichen werden und die Benutzer dadurch einen fairen Teil der Ressourcen erhalten. Die derzeitige FS Implementation hat Details, betreffend Nachrichtenaustausch, der Implementation offengelassen. Dies führt zu einem massiven Austausch an Nachrichten. Diese Arbeit entwickelt logische Nachrichtenflusstopologien, um den Nachrichtenaustausch zu minimieren, und beurteilt verschiedene Implementationsmöglichkeiten zur Eignung für eine einfache Nutzung durch den FS. Zuletzt wird eine Programmierschnittstelle vorgeschlagen, die es möglich macht, verschiedene Nachrichtenflusstopologien, und deren Implementationen, zweckmässig durch den FS zu nutzen. Aufgrund der Programmierschnittstelle und den vorgestellten Implementationsmöglichkeiten kann eine Implementation für den FS vorgenommen werden.



# Abstract

The Fairness Service (FS) achieves fairness among cloud users by continuously gathering information about virtual machines' utilization of multiple resources, aggregating this information to the "heaviness" of users, and adapting virtual machines priorities, such that the heaviness of users is aligned and, therefore, all users have access to a fair share of resources. The current FS implementation has left open multiple implementation details resulting in a massive volume of messages. This thesis develops logical message flow topologies to reduce the message volume and evaluates suitable implementation options for simple use for the FS. Finally, an application programming interface (API) is proposed, making the message flow topologies and the implementation alternatives convenient for FS to use. Based on this, API and the presented topologies implementations can be made for the FS.



# Acknowledgments

I want to thank my supervisors Patrick Gwydion Poullie and Dr. Thomas Bocek for giving me the opportunity to write a thesis in a very interesting and industry-relevant field.

Last but not least, my deepest gratitude to my family and friends for their continuous support and love.





# Contents

<b>Zusammenfassung</b>	<b>i</b>
<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>v</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Thesis Outline . . . . .	2
<b>2 Related Work</b>	<b>3</b>
2.1 Equations of Heaviness . . . . .	3
<b>3 Logical Message Flow Topologies</b>	<b>5</b>
3.1 First Implementation of a Logical Message Flow Topology for the FS . . .	5
3.2 Fully Meshed Topology . . . . .	5
3.3 Ring Topology . . . . .	6
3.4 Star Topology . . . . .	7
3.5 Topology Changes . . . . .	7
3.5.1 Fully Meshed Topology . . . . .	8
3.5.2 Ring Topology . . . . .	9
3.5.3 Star Topology . . . . .	11
3.6 Proposed Pseudocodes . . . . .	11
3.6.1 Fully Meshed Topology . . . . .	12

3.6.2	Ring Topology . . . . .	13
3.6.3	Star Topology . . . . .	15
<b>4</b>	<b>Implementation Options</b>	<b>17</b>
4.1	Definition of Comparison Parameters . . . . .	17
4.1.1	Scalability . . . . .	17
4.1.2	Availability . . . . .	18
4.2	NoSQL Data Stores . . . . .	18
4.2.1	Distributed Document Data Stores . . . . .	19
4.2.2	Distributed Key-Value Stores . . . . .	21
4.3	Distributed Messaging Systems . . . . .	25
4.3.1	AMQP Components . . . . .	26
4.3.2	RabbitMQ . . . . .	30
4.3.3	ZeroMQ . . . . .	31
<b>5</b>	<b>Evaluation</b>	<b>33</b>
5.1	Evaluation Environment . . . . .	33
5.1.1	Vagrant . . . . .	33
5.1.2	Ansible . . . . .	33
5.2	Setting Up a MongoDB Server . . . . .	34
5.2.1	Evaluation of a MongoDB Server . . . . .	34
5.3	Setting Up a MongoDB Cluster . . . . .	34
5.3.1	Evaluation of a MongoDB Cluster . . . . .	35
5.4	Setting Up a Redis Master-Slave Environment . . . . .	36
5.4.1	Evaluation of a Redis Master-Slave Environment . . . . .	36
5.5	RabbitMQ . . . . .	37
5.6	ZeroMQ . . . . .	37
5.7	Combinations . . . . .	39
5.7.1	MongoDB and Redis . . . . .	39

<i>CONTENTS</i>	ix
<b>6 Designing of an API</b>	<b>41</b>
<b>7 Summary and Conclusions</b>	<b>45</b>
7.1 Future Work . . . . .	45
<b>Bibliography</b>	<b>47</b>
<b>Abbreviations</b>	<b>51</b>
<b>List of Figures</b>	<b>53</b>
<b>List of Tables</b>	<b>55</b>
<b>A Installation Guidelines</b>	<b>57</b>
A.1 Windows Preparations . . . . .	57
A.1.1 1. Step - Installing Oracle's VirtualBox . . . . .	57
A.1.2 2. Step - Installing Windows Subsystem for Linux (WSL) . . . . .	57
A.1.3 3. Step - Installing Vagrant . . . . .	58
A.1.4 4. Step - Move files from Windows to WSL . . . . .	58
A.2 Ubuntu Preparations . . . . .	59
A.2.1 1. Step - Installing Oracle's VirtualBox . . . . .	59
A.2.2 2. Step - Installing Vagrant . . . . .	59
A.2.3 3. Step - Installing Ansible . . . . .	59
A.2.4 4. Step - Running Demos . . . . .	59
<b>B Contents of the CD</b>	<b>61</b>



# Chapter 1

## Introduction

This chapter points out the current state, the goals of this thesis, and with which methodologies is worked with.

### 1.1 Motivation

Cloud computing allows for the large-scale sharing of computing resources in a technically and administratively scalable manner. Therefore, a large number of companies (not limited to the information technology sector) have quickly adopted it, thus resulting in high end-user adoption. Although cloud computing has already enabled a plethora of new applications and business models, expertise in exploiting its full potential is still vastly missing [48]. Even though companies are searching for cloud experts, this topic is not always covered in academia. To bridge this gap, this bachelor thesis enables the acquiring of industry-relevant competence by focusing on message exchange among the compute nodes, which hosts virtual machines (VMs) for a user, that constitute a cloud.

The bachelor thesis of Stephan Mannhart [16] extended OpenStack via the nova-fairness service, referred to as the Fairness Service (FS). The FS achieves fairness among cloud users by (i) continuously gathering information about VMs' utilization of multiple resources, (ii) aggregating this information to the "heaviness" of users, or the burden that user put on the cloud, and (iii) adapting VM priorities, such that the heaviness of users is aligned and, therefore, all users have access to a fair share of resources. The Communications Systems Group (CSG) defined the Greediness Metric (GM), which was proven to result in fair allocations and to give the incentive to users to configure the virtual resources (VRs) of their VMs according to the expected load of these, when the GM is deployed as a heaviness metric for the FS.

Although the FS as implemented by Mannhart in [16] was proven to achieve the designated functionality in a brief evaluation, multiple implementation details have been left open. In particular, the message exchange among compute nodes running the FS leaves room for improvement. The goal of this bachelor thesis is to make this improvement possible.

Two essential node types in the OpenStack architecture are the controller node and the

compute nodes. The controller node coordinates the cloud, e.g., by scheduling VMs, and stores essential information for other nodes or administrators to access. Compute nodes perform the actual processing of workloads, i.e., hosting VMs. For this purpose, compute nodes run the OpenStack service nova compute that is part of the OpenStack compute project nova. The FS is implemented as an additional nova service called nova-fairness. Accordingly, an instance of the FS runs on every compute node and has a server counterpart on the controller node. In particular, the server hosts a centralized message broker that relays all messages. Therefore, by default, all information exchanged between compute nodes traverses the controller node. This is not scalable and introduces a single point of failure (although, arguably, the controller node is always a single point of failure). To overcome this drawback, the message exchange between compute nodes (and thus the FS) is decentralized as described in [37]. Unfortunately, this decentralization still results in a message volume that is quadratic in the number of nodes (as every node directly sends a message to every other node) and that is produced periodically.

The goal of this thesis is to develop different alternatives for logical message flow topologies suited for the FS. Different implementation options to allow for compute node communication should be compared, implemented, and tested for use with the FS. Finally, a well-defined API has to be developed, making the message flow topologies and implementation alternatives convenient for the FS to use.

This thesis starts with literature research. Thereafter, a practical implementation is done and concludes with a proposal for a common API.

## 1.2 Thesis Outline

This thesis is divided into the following remaining chapters:

**Chapter 2** summarizes related work done, on which this thesis is based.

**Chapter 3** presents different topologies and presents pseudocodes for possible implementations.

**Chapter 4** discusses potential applications for implementing the presented topologies and discusses their applicability in terms of availability and scalability.

**Chapter 5** evaluates potential applications in terms of their applicability to logical message flow topologies.

**Chapter 6** discusses and provides a common API for the FS to communicate over different topologies.

**Chapter 7** summarizes and concludes this thesis and proposes future work.

# Chapter 2

## Related Work

This thesis relies heavily on the previous studies and work of Stephan Mannhart in his bachelor' thesis [16] and the white paper of Poullie, Mannhart, and Stiller [37]. The work of Poullie et al. laid the theoretical foundation for fair allocation in clouds. They defined and proved several equations that are required to calculate the heaviness of an user. In the white paper of Poullie et al. [37], the theoretical equations were applied with measurable values from the OpenStack cloud, which are important for this thesis and will be explained in the following section.

### 2.1 Equations of Heaviness

A cloud consists of a set of users, often called the tenant, project, or customer in different clouds [31, 18],  $U = \{u_1, u_2, \dots, u_x\}$ ; a set of nodes — in this particular context, nodes are meant as compute nodes,  $N = \{n_1, n_2, \dots, n_y\}$  — and a set of VMs,  $V = \{v_1, v_2, \dots, v_z\}$ . Set  $R = \{r_1, r_2, \dots, r_m\}$  consists of the physical resources (PRs) that nodes provide, which VMs can use, e.g., processing time, memory, etc. In the white paper of Poullie et al. [37] the following four functions were defined, which are essential for the heaviness equations.

$o : V \rightarrow U$  maps a VM to its owner

$a : V \rightarrow N$  maps a VM to the host that accommodates/hosts the VM

$r : V \cup N \cup U \rightarrow \mathbb{R}_{\geq 0}^m$  maps (i) VMs to their VRs, (ii) nodes to their PRs, and (iii) users to their quotas

$l : V \rightarrow \mathbb{R}_{\geq 0}^m$  maps VMs to the loads they impose on the PRs

To calculate the heaviness of VMs, PRs are used as parameters. As PRs are heterogeneous, meaning they are measured in different units, e.g., disk I/Os, processing time in seconds, memory in MB, etc., they have to be normalized with their weighting. This is called the cloud-wide unique normalization vector. To calculate this normalization vector the cloud

resource supply (CRS) is needed. The CRS is the combined PRs of all nodes, which is denoted as the node resource information (NRI), and is calculated by function

$$\text{CRS} := \sum_{n_j \in N} r(n_j).$$

VM endowment calculates the penalty an user gets when the VM's load does not conform with the, for it, reserved PRs. The white paper of Poullie et al. [37] recommended defining the endowment of a VM as the VM's proportional share of the host's PR

$$e(v_i) \mapsto \frac{r(a(v_i))}{\sum_{v_j \in V: a(v_j)=a(v_i)} r(v_j)} \cdot r(v_i).$$

Therefore, the used PRs of the user's VMs during runtime, meaning a specific point in time, must be collected. This is denoted by the function  $l$  and is called the runtime utilization information (RUI).

The heaviness of VMs is defined as

$$H_V(v_i) := s(e(v_i)) + d(v_i).$$

Function  $s : \mathbb{R}_{\geq 0}^m \rightarrow \mathbb{R}_{\geq 0}$  denotes the *static heaviness* as a cost of instantiating a VM. Function  $s$  must strictly increase with each input dimension.

Function  $d : V \rightarrow \mathbb{R}$  is called the *dynamic heaviness* and represents the cost of providing for the VM's load.  $d$  must be chosen, such that  $d(v_i) > -s(e(v_i))$  and the non-minimalistic condition, which is explained in greater detail in section III.C.4 in [37].

To calculate the heaviness of an user compared with other users, the actual user quota, which is defined as the share of the cloud's PRs that is proportional to  $u_i$ 's quota, is needed, and it is calculated by function

$$aq(u_i) \mapsto \frac{\text{CRS}}{\sum_{u_k \in U} r(u_k)} \cdot r(u_i).$$

This results in a vector of users' quotas, whereas the heaviness of an user is scalar. Thus, function  $s' : \mathbb{R}_{\geq 0}^m \rightarrow \mathbb{R}_{\geq 0}$  has to be used to map the vector to a scalar. The recommendation of Poullie et al. [37] was  $s' = s$ . The heaviness of user  $u_i$  "is defined as the sum of  $u_i$ 's VMs' heaviness subtracted by  $u_i$ 's unused quota" [37]:

$$H_U(u_i) := \left( \sum_{v_j \in V: o(v_j)=u_i} H_V(v_j) \right) - s'(aq(u_i)).$$



# Chapter 3

## Logical Message Flow Topologies

This chapter addresses three logical message flow topologies and covers their assets and drawbacks.

### 3.1 First Implementation of a Logical Message Flow Topology for the FS

The first FS implementation of Poullie, Mannhart, and Stiller [38] used the existing centralized messaging system of OpenStack among compute nodes, which uses RabbitMQ. It was replaced with the alternative ZeroMQ [30] to decentralize the information exchange with minimal configuration changes. “In both cases, the message volume is quadratic in the number of compute nodes, as every node sends messages to every other compute node. Therefore, a message scheme was designed, that arranges the information flow among compute nodes as a ring, i.e., every compute node sends messages to and receives messages from exactly one other compute node. The size of the messages send on this ring is linear in the number of users” [38]. Although, details about topology changes were left out.

### 3.2 Fully Meshed Topology

In a fully meshed topology, which can be represented graphically as in Figure 3.1, any node is directly connected to any other node. Thus, a node is able to send messages to everyone without taking a detour through another node. To form a fully meshed topology, each node needs to maintain a list of neighbors in the topology. A node needs to broadcast to anyone in the topology so that the other nodes can update their lists and answer back to the new node to update its list with the other nodes (cf. section 3.5.1.1). Each node collects the RUI of the hosted VMs and applies function  $H_V$  to calculate the heaviness of the VMs. Since the heaviness of users can be calculated in a decentralized manner, each node applies function  $H_U$  to the calculated VM heaviness set, resulting in partial

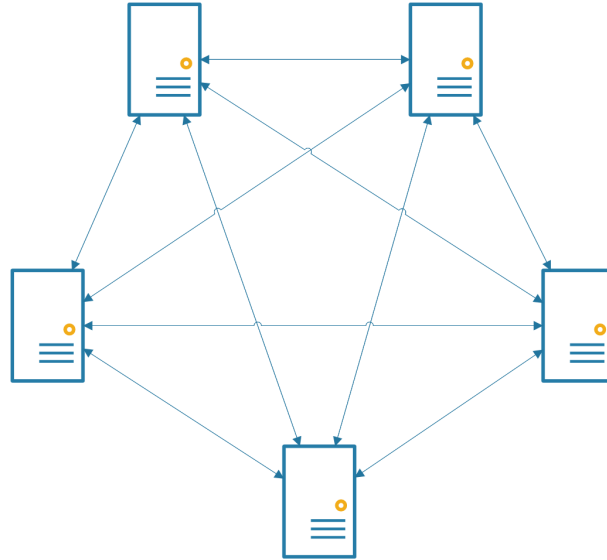


Figure 3.1: Representation of a fully meshed topology.

heaviness of users. Thereafter, the partial user heaviness is sent to the other nodes, where they get aggregated to the final heaviness of users (cf. section 3.3, which applies a similar method).

### 3.3 Ring Topology

In a ring topology, a node has two neighbors, and messages are sent in only one direction along the path. This is shown in Figure 3.2. A known implementation of this topology is the Fiber Distributed Data Interface (FDDI) [2]. To form a ring topology, a beginner, called a master, is necessary. The master is responsible for maintaining the order in a ring. To calculate  $H_U$ , the master will initialize a vector with each entry containing the heaviness of an user and send this vector to its successor. In a first cycle, each node, when receiving the vector from its predecessor, calculates the new heaviness of the user and updates the vector. After one cycle, not every node knows the correct set of  $H_U$  so a second cycle is needed. In this algorithm no set of  $H_V$  is sent to another node. Nonetheless, a redundant set of  $H_V(n_i)$  of node  $n_i$  needs to be stored on another node to

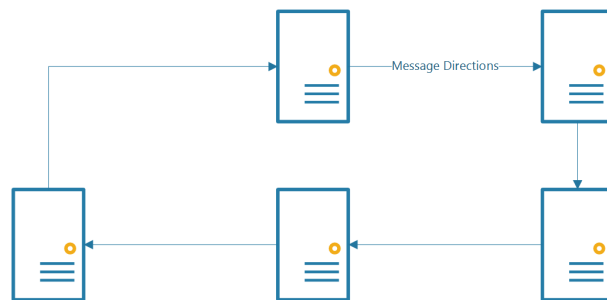


Figure 3.2: Representation of a ring topology.

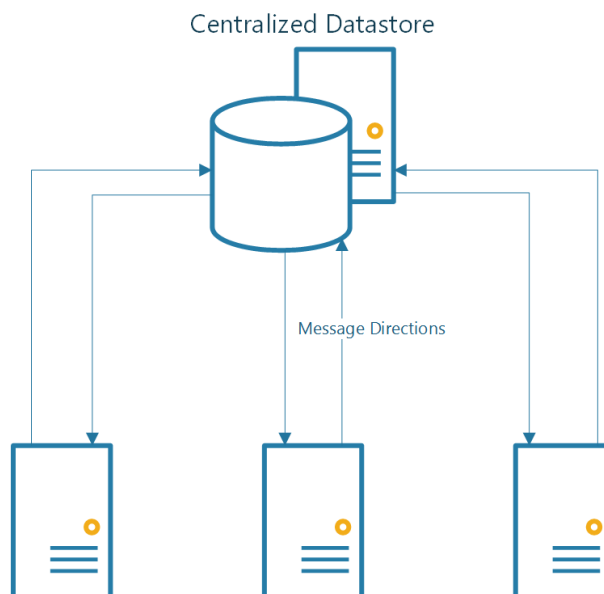


Figure 3.3: Representation of a star topology.

recover from an ungraceful removal, which is discussed in section 3.5.2.2.

## 3.4 Star Topology

A star topology consists of a central node and nodes around this node [49]. It thus looks like a star (cf. Figure 3.3). Most of the computer networks today are in form of this topology, e.g., a switch acts as the central node, and the computers are connected to this switch. To calculate the heaviness of an user, the nodes calculate their partial  $H_U$  sets and send it to the central node, which then aggregates the  $H_U$  sets of all of the compute nodes to calculate the final  $H_U$  set. This can then be sent back to the nodes. The disadvantage of this topology is the higher use of the central node and it must be made sure that it is not a single point of failure. Thus, high availability and fault tolerance is necessary.

## 3.5 Topology Changes

Topology changes occur for a number of reasons. One is adding new nodes to the topology. Another is the removal or a failure of a node, e.g., a hardware or network failure resulting in inaccessible nodes, or the often-forgotten case of rebooting a machine because of a security update. The removal of a node can either happen gracefully or ungracefully. When a node is removed ungracefully, this means the node could not remove itself cleanly from the cluster; it leaves inconsistent data in the cluster. The difficulty lies in the fact that to find out which node failed, a node needs to figure out if it failed or if the remote node failed. The problem resulting from this error state is that the data are inconsistent between some nodes and need recovery. Davidson, Garcia-Molina, and Skeen noted: “One

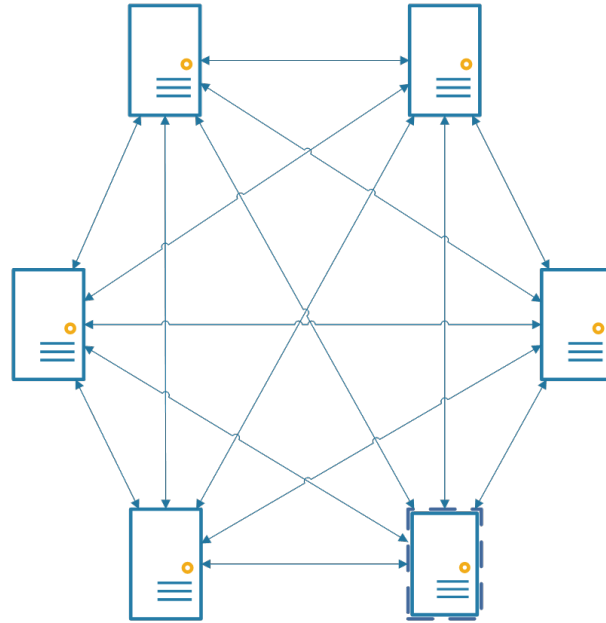


Figure 3.4: Adding a new node to the fully meshed topology.

important aspect of correctness with replicated data is mutual consistency: All copies of the same logical data item must agree on exactly one “current value” for the data item” [6].

### 3.5.1 Fully Meshed Topology

This section discusses operations that should be conducted to add and remove nodes in a fully meshed topology.

#### 3.5.1.1 Adding a Node

To add a new node to the fully meshed topology, the new node broadcasts to every other node that a new node exists in this topology. All remaining nodes need to answer to the new node to let it know where to send the subsequent messages. After this, each node has a new updated list of all participating nodes.

#### 3.5.1.2 Removing a Node

To remove a node gracefully, the to-be-removed node notifies the other nodes that they need to remove the  $H_V$  set of this node, resulting in no inconsistent data. In the case of an ungraceful removal, it gets a bit trickier to recover from the now inconsistent data. The majority of the remaining nodes must agree that a node is unavailable and that its  $H_V$  set can be removed.

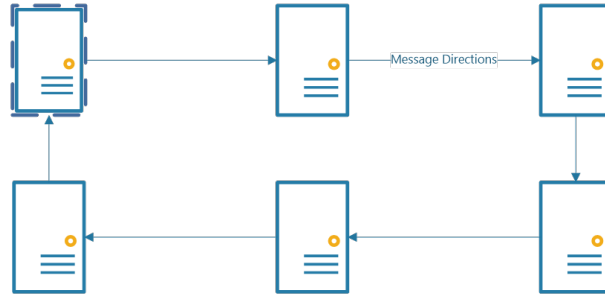


Figure 3.5: Adding a new node to the ring topology.

### 3.5.2 Ring Topology

This section presents the difficulties arising in a ring topology, when changes occur. Nonetheless, possible solutions for these problems are given, too.

#### 3.5.2.1 Adding a Node

To add a new node to the ring topology, the new node broadcasts that a new node is available for the master to pick up. The master needs to rearrange the nodes to maintain the order in the ring, e.g., the last node now needs to send messages to the new node, whereas the new node needs to send its messages to the master. As more nodes are added, the longer the round trip time (RTT) becomes. To mitigate the increasing RTT, a multi-ring topology could be established as shown in Figure 3.6. Establishing a multi-ring topology introduces a higher complexity in terms of reconfiguration and messaging. The masters of the rings need to form a ring to exchange the intermediate aggregated messages of their rings. For example, the master of ring 1 receives an aggregated vector from its ring and passes it on to the master of ring 2. Then, the master of ring 2 introduces the vector it has received from ring 1 into its ring, and so on.

#### 3.5.2.2 Removing a Node

In a graceful removal (numbered messages colored in blue in Fig. 3.7), to-be-removed node  $n_i$  instructs the master node (1) that it will be removed, and it passes the  $\Delta H_U(n_i)$  required to update the  $H_U$  set. After this, the master node acknowledges the removal of node  $n_i$  and can rearrange the order of the nodes (3) and instruct node  $n_{i-1}$  to update the  $H_U$  set accordingly. In the event that node  $n_i$  receives the  $H_U$  set due to a race condition, it sends  $H_U$  unchanged to  $n_{i+1}$  in any case. In this case, the  $H_U$  set will be corrected when it reaches  $n_{i-1}$  again.

When the removal is ungraceful, the  $\Delta H_U(n_i)$  added of the removed node  $n_i$  is needed to recover from the inconsistent data. Therefore, a redundant data set of  $\Delta H_U(n_i)$  needs to be stored, e.g., on node  $n_{(i-1)}$ . With this information, node  $n_{(i-1)}$  can subtract  $\Delta H_U(n_i)$ , recover the  $H_U$  set, and inform the master that node  $n_i$  is unavailable. The master then updates the ring order accordingly, which leads to a needed timeout in the event that a

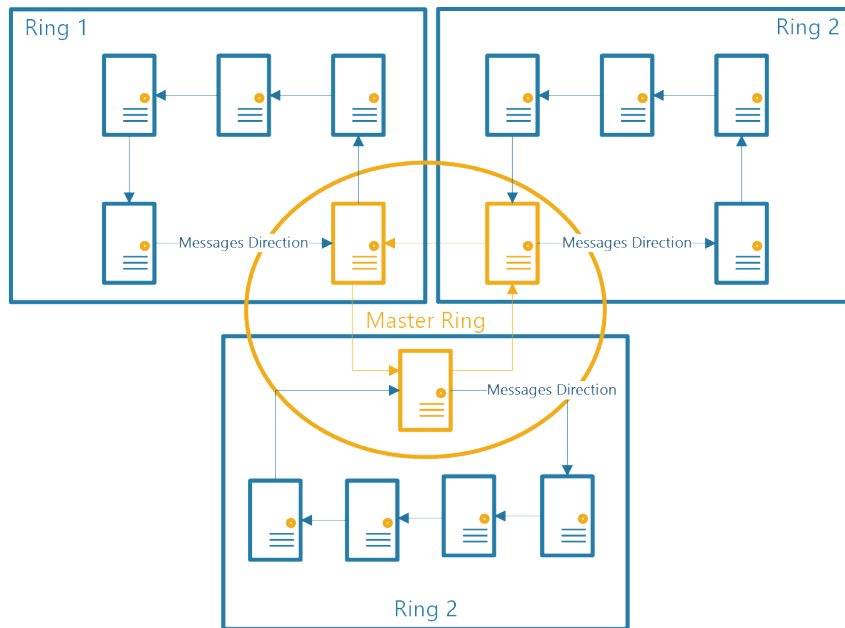


Figure 3.6: Representation of a multi-ring topology.

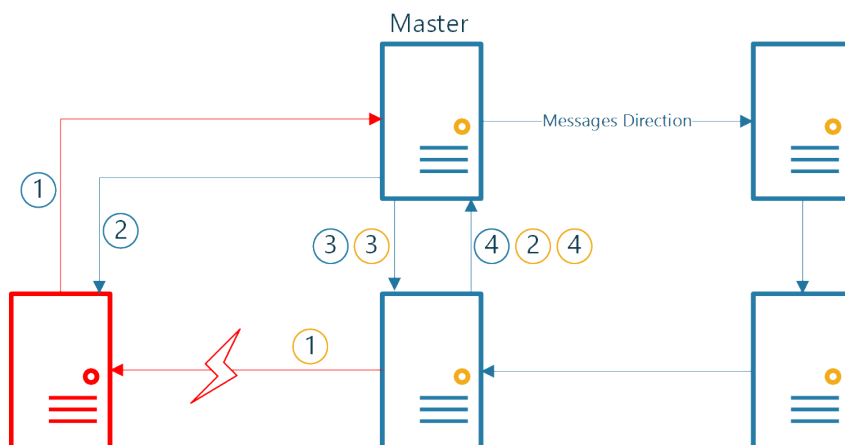


Figure 3.7: Removing a node in a ring topology in two different cases.

node was unwittingly removed, to instruct the master to include it again. The timeout therefore dictates the maximum RTT and consequently the maximum size of a ring. This becomes even more complex when it is extended to a multi-ring topology.

### 3.5.3 Star Topology

This section shows the simplicity of using this topology in the event of changes.

#### 3.5.3.1 Adding a Node

To add a node in a star topology, only a connection between the central node and the new node has to be established. Other nodes that are already in the topology do not need to know that there is a new node. This makes it easy to add a new node in terms of configuration complexity.

#### 3.5.3.2 Removing a Node

In a star topology, a combination of the ring and broadcast topology could be used to remove a node. In the event of a graceful removal, to-be-removed node  $n_i$  announces it to the central node, which then removes the partial  $H_U(n_i)$  set and updates the new heaviness of users. If expire times  $t_e$  are set for the partial  $H_U$  sets and removed when  $t_{now} \geq t_e$ , even an ungraceful removal leads to a recovery of the inconsistent data. Furthermore, the removal of a node could be broken down to just allow the partial  $H_U$  sets to always expire, thus reducing the complexity even more. Another option instead of a timeout would be the central node's polling the liveness (heartbeat) of the compute nodes. If one heartbeat fails, it could remove the failed node.

## 3.6 Proposed Pseudocodes

The current implementation [16] uses interval  $\mu$  in seconds to calculate the new  $H_U$  sets. The proposed pseudocodes have introduced timeout  $\theta$  to handle failed or non-responsive nodes. In consequence, if too small interval  $\mu$  was chosen, timeout  $\theta$  may extend the calculation interval. Furthermore, it can be differentiated between an accurate and an inaccurate calculation. The accurate calculation has the goal of resulting in an  $H_U$  set, which is identical on all nodes  $n \in N^f$  within the same calculation cycle. This can be broken down to two phases:

1. The gather phase.
2. The calculation phase.

During the gather phase, the cluster agrees on a common set of shared variables needed for calculation. This means the cluster first has to figure out which nodes belong to the set  $N^f$  for the current calculation cycle. Next, the CRS value has to be determined, and every node in  $N^f$  must agree on it. After this, the calculation phase begins, proceeding with exchanging the variables needed to calculate the  $H_U$  set.

The inaccurate calculation leads to different  $H_U$  sets among nodes  $n \in N^f$  but should converge to the same  $H_U$  over time.

The reason for having two separate implementations is the different handling in the case of a failing node in the  $N^f$  set, as this has an impact on the CRS, or the fact that node  $n_i$  has not finished calculating its  $H_V(n_i)$  set and that interval  $\mu$  has been given priority, resulting in the removal of the NRI of node  $n_i$  from CRS or in the use of an older  $H_V(n_i)$  set for calculation.

### 3.6.1 Fully Meshed Topology

```

set timeout  $\theta$  in seconds
phase1:
  collect participating nodes in  $N^f$ 
  while NRI of some node in  $N^f$  is missing:
    send own NRI to nodes of which NRI is missing
    use NRIs to calculate CRS and normalization vector
phase2:
  every  $\mu$  seconds:
    collect RUI of all VMs hosted by  $n_i$ 
    apply  $H_V$  to collected RUI in order to calculate
      heaviness of all VMs hosted by  $n_i$ 
    apply  $H_U$  to calculate the heaviness of all  $u \in U$ 
    send this  $H_U$  heaviness set to all  $n \in N^f - \{n_i\}$ 
    wait to receive heaviness set from all  $n \in N^f - \{n_i\}$  or
      timeout  $\theta$  expired
    if timeout  $\theta$  expired
      send restart message to all  $n \in N^f$ 
      go to phase1
    aggregate  $H_U$  sets to calculate the heaviness of all  $u \in U$ 
    for every VM  $v$  hosted by  $n_i$ :
      set priorities of  $v$  according to  $H_V(v)$  and  $H_U(\text{host}(v))$ 

```

Listing 3.1: Steps of the FS accurate calculation running on node  $n_i$  in a fully meshed topology

The pseudocode in Listing 3.1 stops the calculation as soon as a timeout occurs on any node. This leads to an abort as soon as any node has a failure, and with a higher number of nodes, it is certain that at any given time a node has a problem, rendering this method is not suitable at all.



```

set timeout  $\theta$  in seconds
phase1:
  collect participating nodes in  $N^f$ 
  while NRI of some node in  $N^f$  is missing:
    send own NRI to nodes of which NRI is missing
  use NRIs to calculate CRS and normalization vector
phase2:
  every  $\mu$  seconds:
    collect RUI of all VMs hosted by  $n_i$ 
    apply  $H_V$  to collected RUI in order to calculate
      heaviness of all VMs hosted by  $n_i$ 
    apply  $H_U$  to calculate the heaviness of all  $u \in U$ 
    send this  $H_U$  heaviness set to all  $n \in N^f - \{n_i\}$ 
    wait to receive heaviness set from all  $n \in N^f - \{n_i\}$  or
      timeout  $\theta$  expired
    aggregate  $H_U$  sets to calculate the heaviness of all  $u \in U$ 
    for every VM  $v$  hosted by  $n_i$ :
      set priorities of  $v$  according to  $H_V(v)$  and  $H_U(\text{host}(v))$ 
    if timeout  $\theta$  expired
      go to phase1

```

Listing 3.2: Steps of the FS inaccurate calculation running on node  $n_i$  in a fully meshed topology

The inaccurate calculation in Listing 3.2 is basically the same as the accurate calculation, with the difference being that it does not stop when a timeout occurs. Still, after a timeout, it has to gather  $N^f$  to detect topology changes.

In both cases, the message volume needed to calculate the CRS is  $|N| \cdot (|N| - 1) \cdot |R| \Rightarrow \mathcal{O}(|N|^2 \cdot |R|)$ , as the NRI of node  $n_i$  has to be sent to all remaining nodes  $n \in N^f - \{n_i\}$ . To calculate the heaviness of user  $u_i$ , all nodes must know  $s'(aq(u_i))$ , which results in a message volume of  $\mathcal{O}(|N| \cdot |U|)$ . The heaviness calculations run in the same manner as the CRS calculation. Each node has to send every other node the heaviness of the users it calculated, resulting in a message volume of  $|N| \cdot (|N| - 1) \cdot |U| \Rightarrow \mathcal{O}(|N|^2 \cdot |U|)$ , which Poullie et al. [37] already explained. Thus, the summed message volume is  $\mathcal{O}(|N|^2 \cdot |R| + |N| \cdot |U| + |N|^2 \cdot |U|)$ .

### 3.6.2 Ring Topology

```

set timeout  $\theta$  greater than  $\mu$  in seconds
phase1:
  figure out who the successor node  $n_{i+1}$  is
  wait to receive CRS and normalization vector
    from predecessor node  $n_{i-1}$ 
  use CRS from predecessor and own NRI to calculate new CRS
    and normalization vector
  send new CRS and normalization vector to successor node
  wait to receive CRS and normalization vector

```

from predecessor node  $n_{i-1}$  and send it to successor

phase2:

- if  $n_i$  is the master node:
  - start cycle timer  $\phi$
  - every  $\mu$  seconds and wait for  $H_U$  set received or timeout  $\theta$  expired:
    - if timeout  $\theta$  expired:
      - send restart message to all  $n \in N^f$
      - go to phase1
  - remove previous added  $H_U$  set from the received  $H_U$  set
  - collect RUI of all VMs hosted by  $n_i$
  - apply  $H_V$  to collected RUI in order to calculate heaviness of all VMs hosted by  $n_i$
  - apply  $H_U$  to calculate the heaviness of all  $u \in U$
  - send this  $H_U$  heaviness set to the successor
  - for every VM  $v$  hosted by  $n_i$ :
    - set priorities of  $v$  according to  $H_V(v)$  and  $H_U(\text{host}(v))$
  - wait for  $H_U$  set received or timeout  $\theta$  expired:
    - if timeout  $\theta$  expired:
      - send restart message to all  $n \in N^f$
      - go to phase1
  - stop cycle timer  $\phi$  and compare it to  $\theta$ , adapt  $\theta$  if needed, and send  $\theta$  to the successor
  - if timeout expired received from any node  $n \in N^f$ :
    - send restart message to all  $n \in N^f$
    - go to phase1
  - go to phase2
- collect RUI of all VMs hosted by  $n_i$
- apply  $H_V$  to collected RUI in order to calculate heaviness of all VMs hosted by  $n_i$
- apply  $H_U$  to calculate the heaviness of all  $u \in U$
- wait for  $H_U$  set received or timeout  $\theta$  expired:
  - if timeout  $\theta$  expired:
    - send timeout expired to master
- remove previous added  $H_U$  set from the received  $H_U$  set
- aggregate received  $H_U$  set with own  $H_U$  set
- send this  $H_U$  heaviness set to the successor
- for every VM  $v$  hosted by  $n_i$ :
  - set priorities of  $v$  according to  $H_V(v)$  and  $H_U(\text{host}(v))$
- if new  $\theta$  received:
  - adapt  $\theta$  to new value and send it to the successor

```
go to phase2
```

Listing 3.3: Steps of the FS inaccurate calculation running on node  $n_i$  in a ring topology.

As discussed in section 3.3, the ring topology needs a master. The idea behind this topology is to combine all  $H_U$  sets from the predecessor nodes, which finally incorporates all  $H_V$  sets at the last node,  $n \geq n_i, \forall n \in N^f$ , assuming that the master is the first node,  $n_1 \in N^f$ . The main difficulty in this topology lies in the adding and removal of nodes, as the master has to inform particular nodes about topology changes. Not only that, but also the CRS changes for any topology change. Thus, it is recommended to make topology changes at the beginning or the end of a phase and to start all over with *phase1*. The same recommendation applies for changes in the  $U$  and  $V$  sets, too. Changes in the  $U$  and  $V$  sets also increase the RTT, thus increasing the possibility of a timeout. Therefore, it is recommended to monitor the RTT, which is denoted as  $\phi$  in the pseudocode, and to increase or decrease the timeout  $\theta$ , e.g., with an exponential backoff. The exponential backoff can also be used at *phase1* when  $\theta$  is too low or is unknown at the beginning. This pseudocode does not handle master failover, in case the master disappears, and is subject to future work.

To calculate the CRS, a message volume of  $2 \cdot |N| \cdot |R| \Rightarrow \mathcal{O}(|N| \cdot |R|)$  is needed, as after the first cycle, the full CRS is known only to the last node. Thus, a second cycle is needed. The same applies to  $s'(aq(u_i))$ , which results in a message volume of  $\mathcal{O}(|N| \cdot |U|)$ . With the proposed pseudocode, a linear message volume in terms of the user and the nodes is used to update the heaviness of users  $\mathcal{O}(|N| \cdot |U|)$ . Thus, the summed message volume is  $\mathcal{O}(|N| \cdot |R| + 2 \cdot (|N| \cdot |U|))$ .

### 3.6.3 Star Topology

```
phase1:
```

```
    send NRI to central node
```

```
phase2:
```

```
    every  $\mu$  seconds:
```

```
        collect RUI of all VMs hosted by  $n_i$ 
```

```
        apply  $H_V$  to collected RUI in order to calculate
            heaviness of all VMs hosted by  $n_i$ 
```

```
        send this heaviness set to the central node
```

```
        get  $H_U$  from the central node
```

```
        for every VM  $v$  hosted by  $n_i$ :
```

```
            set priorities of  $v$  according to  $H_V(v)$  and  $H_U(\text{host}(v))$ 
```

```
        go to phase2
```

Listing 3.4: Steps of the FS accurate calculation running on node  $n_i$  in a star topology

In a star topology, a node registers its NRI at the central node. As the central node does all of the calculation of the user heaviness, CRS and  $s'(aq(u_i))$  do not need to be broadcasted to any node. The registration at the central node is a single message per node; thus,  $\mathcal{O}(|N| \cdot |R|)$ . The heaviness of VMs generates a message volume of  $\mathcal{O}(|N| \cdot |V|)$ . The message volume to get the heaviness of users amounts to  $\mathcal{O}(|N| \cdot |U|)$ . The total

message volume adds up to  $\mathcal{O}(|N| \cdot |R| + |N| \cdot |V| + |N| \cdot |U|)$ . As  $|V| \gg |U|$ . Because more VMs than users are typically present, sending the CRS and  $s'$  ( $aq(u_i)$ ) to the nodes is usually feasible. The nodes, then apply  $H_U$  on their behalf and send this set back to the central node, where it is finally aggregated. This would reduce the message volume to  $\mathcal{O}(2 \cdot (|N| \cdot |R|) + 2 \cdot (|N| \cdot |U|))$  and result in the pseudocode in Listing 3.5 and Listing 3.6.

```

set timeout  $\theta$  greater than  $\mu$  in seconds
phase1:
    send NRI to central node
phase2:
    every  $\mu$  seconds:
        collect RUI of all VMs hosted by  $n_i$ 
        apply  $H_V$  to collected RUI in order to calculate
            heaviness of all VMs hosted by  $n_i$ 
        wait to receive CRS or timeout  $\theta$  expired
        if timeout  $\theta$  expired:
            go to phase1
        apply  $H_U$  to calculate the partial heaviness of users
        send this heaviness set to the central node
        wait to receive final  $H_U$  from the central node
            or timeout  $\theta$  expired
        if timeout  $\theta$  expired:
            go to phase1
        for every VM  $v$  hosted by  $n_i$ :
            set priorities of  $v$  according to  $H_V(v)$  and  $H_U(\text{host}(v))$ 
        go to phase 2

```

Listing 3.5: Steps of the FS accurate calculation running on node  $n_i$  in a star topology

```

set timeout  $\theta$  greater than  $\mu$  in seconds
phase1:
    wait to receive NRI
phase2:
    every  $\mu$  seconds:
        send CRS to nodes
        wait to receive all  $H_U$  sets of  $N^f$  or timeout  $\theta$  expired:
        aggregate the  $H_U$  sets to a final  $H_U$  set and
            send it to nodes
        remove NRI of timed out nodes

```

Listing 3.6: Steps of the FS accurate calculation running on the central node in a star topology

The central node aggregates all  $H_U$  sets to a final one. If a node does not send a set, it is considered not to be participating anymore. Thus, its NRI is removed from the CRS for the next cycle. A non-participating node, therefore, must first register itself again before the next cycle takes place.

# Chapter 4

## Implementation Options

In this chapter, different implementation options are evaluated and compared with one another, namely:

1. NoSQL data stores
  - Distributed document stores, such as MongoDB
  - Distributed key-value stores, such as Redis and etcd
2. Distributed messaging systems, such as RabbitMQ and ZeroMQ

### 4.1 Definition of Comparison Parameters

The comparison relies on two parameters, namely scalability (cf. section 4.1.1) and availability (cf. section 4.1.2).

#### 4.1.1 Scalability

Scalability is not easy to define [13]. André Bondi [3] distinguished scalability into four groups:

- Load scalability
- Space scalability
- Space-time scalability
- Structural scalability

In addition, Margaret Rouse defined scalability as “ [...] the ability of a computer application or product (hardware or software) to continue to function well when it (or its context) is changed in size or volume in order to meet a[sic!] user need” [47].

This thesis uses load scalability as a comparison parameter.

#### 4.1.1.1 Load Scalability with Vertical or Horizontal Scaling

Load scalability can be performed in two ways:

- Vertical scaling, or scale-up
- Horizontal scaling, or scale-out

Vertical scaling works such a way that more of a resource is added to a single node to keep an application working on a higher workload, for example, adding more CPUs, replacing the CPU with a more powerful one, memory, disk drives, etc. Thus, it is limited in a technological way.

Horizontal scaling works in such a way that more nodes are added and the workload is distributed across these nodes. Thus, the hardware used is combined to fulfill the workload.

In this thesis horizontal scaling is taken into account in comparing the implementation options.

#### 4.1.2 Availability

Availability, or high availability, is the ability to keep a system running for a long period of time [46]. As software or systems regularly fail, the planning of backups and failovers has to be conducted. In this thesis, software is compared, and thus, the ability the software provides to keep it high available is reviewed.

## 4.2 NoSQL Data Stores

Modern applications are now meeting requirements that were not met with traditional relational databases, thus leading to a new development of “not only SQL (NoSQL)” data stores. NoSQL data stores provide a flexible data model, where fields can be added on the fly without downtime, and where the fields do not have to be the same across records as in relational databases. This makes it easy to store and combine the data of any structure, such as document, graph, key-value, or wide-column. “NoSQL databases were all built with a focus on scalability, so they all include some form of sharding or partitioning” [23] (cf. section 4.2.0.1 about sharding and partitioning). With these possibilities, NoSQL data stores can run on many nodes and scale across on-premises as well as in the cloud, and some even replicate across the data centers, thus allowing horizontal scaling and delivering a higher throughput and lower latency than relational databases do [23], [15]. This does not come without a drawback, however. The structured query language (SQL) is a standard because “all relational databases have the same concept of storing data in tables” [36]. Although a switch from one relational database to another still requires a little change, “it is much easier than switching between two different NoSQL data stores.

Because each NoSQL data store has unique aspects in both how its data is stored as well as how different bits of data relate to each other, no single API manages them all. When embracing a new NoSQL data store, the developer must invest time and effort to learn the new query language as well as the consistency semantics” [36].

#### 4.2.0.1 Sharding and Partitioning

Sharding or partitioning is a method of splitting data into multiple instances of a data store. These instances can be on the same node, but in terms of scalability, partitioning over multiple nodes makes more sense. In this way, the size of the data stored can be higher than the memory or the disk space of a single node. Without this splitting, the limitation would be the vertical scalability (cf. Section 4.1.1.1) of a node. Furthermore, the load in terms of computation and network bandwidth is shared across the cluster-composing nodes. This technique facilitates horizontal scaling. The exact implementation of partitioning differs from one data store to another, which will be discussed in the following sections.

### 4.2.1 Distributed Document Data Stores

A document data store is a class of a NoSQL database. This class is discussed in the next section.

#### 4.2.1.1 MongoDB

MongoDB, Inc. defined MongoDB as: “MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need” [28]. MongoDB allows multiple storage engine to be mixed within a single deployment, namely WiredTiger, MMAPv1, and the In-Memory Storage Engine. “The storage engine is the component of the database that is responsible for managing how data is stored, both in memory and on disk. MongoDB supports multiple storage engines, as different engines perform better for specific workloads. Choosing the appropriate storage engine for your use case can significantly impact the performance of your applications” [27]. Comparing these storage engines may lead to an improvement and is subject to future work. Traditional relational databases often need to have separate instances for different storage engines to meet data requirements.

MongoDB supports several data models, such as documents; key-value pairs; flat, table like structures; and objects with deeply nested arrays and sub-documents [23]. In this thesis, only MongoDB version 3.4 and the document data model will be considered, because for the other data models specific software will be evaluated. Although, in the evaluation of MongoDB in chapter 5, version 2.6 of MongoDB is deployed. This version suffices to prove that MongoDB is high available and fault tolerance, that still holds true for version 3.6.

**Data as Documents** In MongoDB documents with a similar structure are organized into collections, which would be a table in a relational database and documents would be a row. A document contains fields, which could be represented as columns. Although relational databases spread a record across many tables due to normalization. “Database Normalization, or simply normalization, is the process of organizing the columns (attributes) and tables (relations) of a relational database to reduce data redundancy and improve data integrity” [50]. Typically, MongoDB stores all data for a record in a single document [23]. These records are stored in a binary representation of the JavaScript Object Notation (JSON) called the Binary JavaScript Object Notation (BSON). Therefore, a document looks like a JavaScript object in Listing 4.1.

```
{
  [ {
    tenant_id: "f34d8f71-7003-4280-a42f-6318d1a4af34" ,
    node_id: "b0ed0a61-a0b0-4e9f-b755-340ecc6a98c3" ,
    instance_id: "078f97cf-19e0-440d-aa5c-1234a75a57d3" ,
    name: "instance-000005a2" ,
    heaviness: "24"
  }, {
    tenant_id: "f34d8f71-7003-4280-a42f-6318d1a4af34" ,
    node_id: "b0ed0a61-a0b0-4e9f-b755-340ecc6a98c3" ,
    instance_id: "11670529-3cd8-47d3-bda9-2ded5738424f" ,
    name: "instance-000006e1" ,
    heaviness: "56"
  } ]
}
```

Listing 4.1: Possible representation of two documents for use with MongoDB

**MongoDB Components** MongoDB consists of several components, of which *mongod*, *mongos*, and *mongo* are used in this thesis. These three components are called the *core processes* [24].

The process responsible for handling data requests, data access management, and background management operations is *mongod*, which is the primary daemon process and therefore the core database process [22].

*Mongos*, which is short for “MongoDB Shard”, is responsible for routing queries in a MongoDB shard configuration to the shards containing the requested documents [25].

MongoDB can be interacted with a JavaScript shell interface called *mongo*. It allows one to “test queries and operations directly with the database” and is intended for system administrators or developers [20].

**Scalability** Horizontal scalability in MongoDB is supported through sharding and its responsible process *mongos*, and in a limited way, it provides load scalability with replication by redirecting some clients to read from secondaries (cf. Section 4.2.1.1). A sharded cluster in MongoDB consists of three components:



**shard** Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set (cf. Section 4.2.1.1).

**mongos** The *mongos* acts as a query router, providing an interface between client applications and the sharded cluster.

**config servers** Config servers store metadata and configuration settings for the cluster.

MongoDB shards the collections in a database, which means the documents are placed on different nodes.

**Availability** MongoDB supports high availability and fault tolerance with a replication mechanism. The replication is done with a group of *mongod* processes called a replica set, which maintains the same data and thus provides data redundancy. “A replica set in MongoDB is a group of *mongod* processes that provide redundancy and high availability. The members of a replica set are:

**Primary** The primary receives all write operations.

**Secondaries** Secondaries replicate operations from the primary to maintain an identical data set. [...]” [21].

**Arbiter** Arbiters do not have copies of the data and are optional. An arbiter maintains a quorum in a replica set by responding to heartbeat and election requests from other replica set members.

“The minimum recommended configuration for a replica set is a three member replica set with three data-bearing members: one primary and two secondary members” [21]. Another option would be to exchange a secondary with an arbiter, resulting in a two data-bearing member cluster, “but replica sets with at least three data-bearing members offer better redundancy” [21] and thus is recommended.

## 4.2.2 Distributed Key-Value Stores

A Key-Value store stores values with a key, as the name implies. This type of database uses a map, dictionary, or an associative array as the data model. Each key is associated with a value in a collection, hence the term “key-value pair”. The key can be, depending on the implementation, a string or hash, or the database may support binary key files, such as images or other Binary Large Objects (BLOBs). Key-value stores generally do not have a query language. They provide commands, such as *get*, *put*, and *delete* to retrieve, store, and delete data [1].

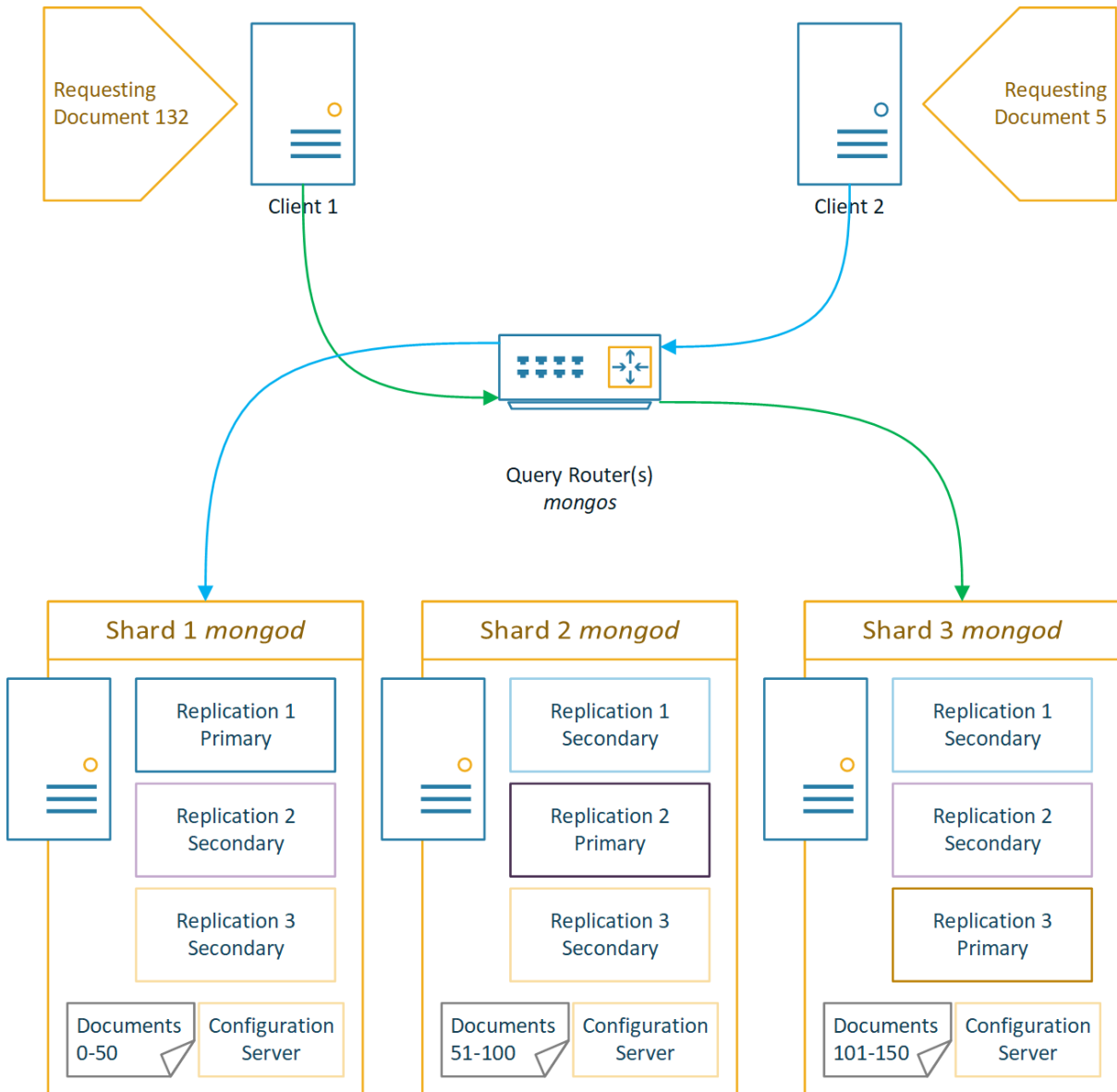


Figure 4.1: Architecture of a three node sharded MongoDB cluster with replica sets and the retrieval process for high availability, fault tolerance, and horizontal scalability.

### 4.2.2.1 Redis

“Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker. It supports data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyperloglogs and geospatial indexes with radius queries. Redis has built-in replication, Lua scripting, LRU eviction, transactions and different levels of on-disk persistence, and provides high availability via Redis Sentinel and automatic partitioning with Redis Cluster” [42]. Redis is a key-value data store with the ability to store complex data structures as a value. Redis stores all data with an in-memory dataset. In addition, the data can be saved on the disk drive at particular points in time or by persisting commands to a log.

**Redis Components** To provide high availability, fault tolerance, and horizontal scalability, Redis consists of:

**Redis** Redis is the main process that stores the data.

**Redis Sentinel** *Redis Sentinel* provides high availability for Redis.

**Redis Cluster** *Redis Cluster* shards automatically data across Redis instances.

**Special Features of Redis** Redis supports expiration for the automatic invalidation of keys, or publish-subscribe for clients subscribing to a key (cf. section 4.3).

**Scalability** Horizontal scalability is supported through *Redis Cluster*. Redis Cluster shards the keys so that a key is a part of a so-called *hash slot*. In a Redis Cluster are 16384 hash slots. To compute the hash slot of a key, the 16-bit cyclic redundancy check (CRC) value of the key is taken modulo 16384. A node in a Redis Cluster manages only a subset of the hash slots. For example, a three-node cluster manages the hash slots accordingly:

- Node 1 manages slots from 0 to 5500.
- Node 2 manages slots from 5501 to 11000.
- Node 3 manages slots from 11001 to 16383.

To add a node to the cluster only some hash slots from the other nodes have to be shifted to the new node. The same applies if a node will be removed from the cluster [43].

**Availability** To ensure availability, Redis uses *Redis Sentinel*. Sentinel constantly checks if master and slave instances are working as expected. If something is wrong with a monitored Redis instance, Sentinel is also able to send notifications to various recipients. If something happens with the master, Sentinel starts a failover process. In this process a slave is promoted to a master, whereas the remaining slaves are reconfigured to use the new master as a replication source. Sentinel also acts as a configuration provider. It provides information for clients to discover the address of the current Redis master. If a failover occurs, Sentinel sends the new address to the clients for reconfiguration [45]. To ensure fault tolerance, Redis has a replication mechanism built into Redis instances, which must be configured accordingly to replicate data. The replication is done asynchronously in an one-master-N-slave scheme. However, that in the event of a master failure, data may be lost. The reason for this is that the master acknowledges a writing to the data store before the slaves have processed the change:

1. Client writes to Redis master
2. Master acknowledges writing
3. Master replicates with slaves

The reason for doing so is speed, as the master does not have to wait for the acknowledgements of the slaves in the trade-off of consistency.

#### 4.2.2.2 etcd

“etcd is an open-source distributed key value store that provides shared configuration and service discovery for Container Linux clusters. etcd runs on each machine in a cluster and gracefully handles leader election during network partitions and the loss of the current leader” [5] (cf. section 4.2.2.2 for Linux containers). As with Redis, applications “write” values and are associated with a key in the data store.

**Special Features of etcd** Besides the basic key-value data store features, such as write, read, and delete keys, etcd supports nested keys with a tree-like structure, which is referred to *directories*, as shown in Listing 4.2.

```

/users
+--+ f34d8f71-7003-4280-a42f-6318d1a4af34
  +--+ 078f97cf-19e0-440d-aa5c-1234a75a57d3
    | +-- heaviness: "24"
    | +-- name: "instance-000005a2"
    | +-- node_id: "b0ed0a61-a0b0-4e9f-b755-340ecc6a98c3"
    |
  +--+ 11670529-3cd8-47d3-bda9-2ded5738424f
    +-- heaviness: "56"
    +-- name: "instance-000006e1"

```

```
+— node_id: "b0ed0a61-a0b0-4e9f-b755-340ecc6a98c3"
```

Listing 4.2: Tree representation of a nested etcd key.

Furthermore, etcd supports the expiration of keys, which is called time to live (TTL). Additionally, etcd allows *watching* for changes of values with notification to interested parties.

**Linux Containers** Containers provides an option for partitioning PRs into isolated groups. Another possibility for sharing PRs is virtualization, but in contrast to virtualization, containers do not have full operating systems (OSs). Containers share the same kernel and run instructions native on the CPU without interpretation or emulation. This gives an application the illusion of running on a separate machine. Sharing resources, while providing isolation, leads to lower overhead compared with true virtualization [12].

**Scalability** etcd does not provide a means for horizontal scalability.

**Availability** etcd is designed to be high available and fault tolerant. It does so with replicating the data across the cluster members. Therefore, a higher number of members decreases the write performance. Although, the read operations are load balanced across the cluster.

## 4.3 Distributed Messaging Systems

Messaging systems are generally responsible for message routing, message queuing, and message passing between the threads of a process (in-process communication), between programs on the same computer, or even over a network (inter-process communication). Using messaging systems promotes the loose coupling of processes through the asynchronicity of messages, as the sender hands over a message to the message system and advances its program, and the message system makes sure that the message will be handed out to a receiver. Sometimes the message can also be transformed to transfer messages between systems with which it would be impossible to communicate by adapting the message before delivering. The transformation therefore requires intelligence on how to adapt messages and is found only in message-oriented middlewares [52]. By providing the queuing of messages, the communication can be independent timewise; neither the sender nor the receiver has to be connected at the same time, thus providing resilience for intermittent network connectivity between processes. The main process for managing a message system is called *message broker*. In a message system, multiple message brokers can be implemented for increasing throughput via load balancing or high availability. As soon as there are multiple message brokers, routing comes into play. Sometimes the routing logic is implemented in the messaging system itself, and sometimes the sender and receiver have to provide information, or both.

An Organization for the Advancement of Structured Information Standards (OASIS) standardized implementation is the Advanced Message Queuing Protocol (AMQP) version 1.0 [14], but as RabbitMQ at the time of writing only stably implements AMQP version 0.9.1, the following sections will discuss the concepts that RabbitMQ supports. The standardization ensures that client interactions are language independent and ensures interoperability between AMQP implementing brokers. The AMQP provides different messaging interactions as follows:

**Request-response pattern** The request-response pattern can be compared to a client-server structure, where a client requests something in particular, for example, a parametrized request, from the server and the server has to respond exactly to this client.

**Store-and-forward technique** Store-and-forward is used when the publisher and subscriber are not connected to the message queue within the same time domain. The publisher can publish a message to a queue, without having the subscribers online. When the subscriber comes online, it can fetch the message from the queue. To fetch stored messages from the queue, the publisher need not to be online.

**Publish-subscribe pattern** A general publish-subscribe system has multiple publishers and subscribers. A process can be a publisher and a subscriber at the same time. The party who provides messages is the publisher. A publisher provides messages for a particular topic. The consumer of these messages is called the subscriber, who subscribes for a particular topic that he or she wants, to receive as soon the messages are published. The publisher does not know which consumers are interested in a topic.

### 4.3.1 AMQP Components

The AMQP concept, which can be represented graphically as in Figure 4.2, allots the deliverance of messages first to an *exchange*. At the exchange messages are routed or copied to one or more queues. The messages are stored in the queue until a consumer fetches them, or they are pushed to the consumer depending on how the consumer has registered himself or herself with the queue. The AMQP allows an administrator or client application to define exchanges and queues, which enables the dynamic creation of queues and exchanges to meet business requirements. The AMQP defines four message exchanges, which are illustrated in the following sections.

#### 4.3.1.1 Direct Exchange

In a direct exchange, messages are delivered to queues based on the message *routing key* (cf. Figure 4.3).

To establish a direct exchange, a queue has to bind to the exchange with routing key  $K$ . When a new message arrives at the direct exchange while routing key  $R$  is being held, the exchange routes it to the queue if  $K = R$  [32].

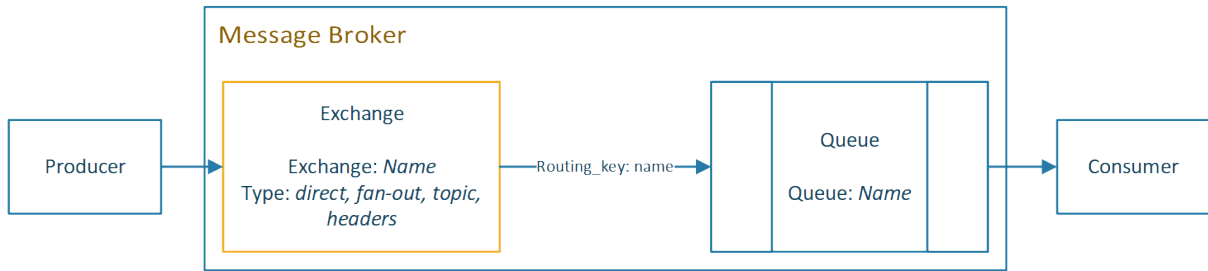


Figure 4.2: Concept of message acquisition, routing, queuing, and delivering in the AMQP.

The direct message exchange is often used to distribute tasks in a round robin manner. Therefore, messages are load balanced among the consumers and not the queues.

The direct exchange has a special case called the *default direct exchange*. The default direct exchange is when a queue binds itself to an exchange without a routing key. In that case, the exchange routes messages with the routing key matching the queue name.

#### 4.3.1.2 Fan-out Exchange

In a fan-out exchange, messages are routed to all bound queues, ignoring the routing key. This exchange is predestined to broadcast messages, for example, to update the configuration, state, or logs in a distributed application. This is graphically represented in Figure 4.4.

#### 4.3.1.3 Topic Exchange

In a topic exchange, messages are routed to bound queues matching the routing key or a pattern of it. The topic exchange is used when the need for a publish-subscribe pattern arises or when messages must be multi-casted.

In Figure 4.5, three different routing key patterns are defined for routing messages to the specific queues:

- `#` matches all routing keys.

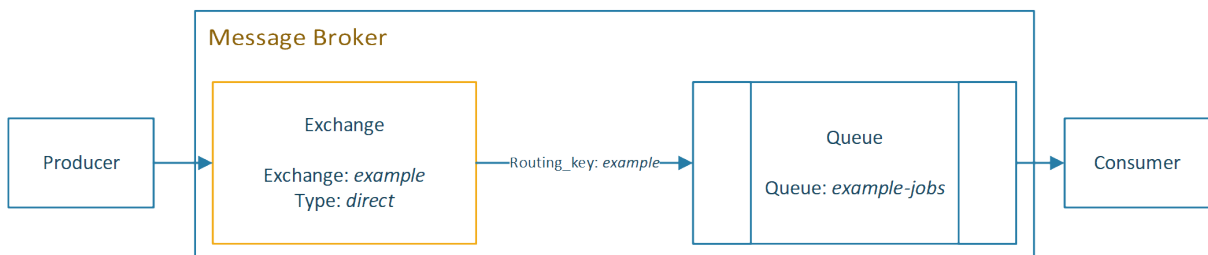


Figure 4.3: A direct exchange called *example* delivers messages with routing key *K example* to the bound queue called *example-jobs*.

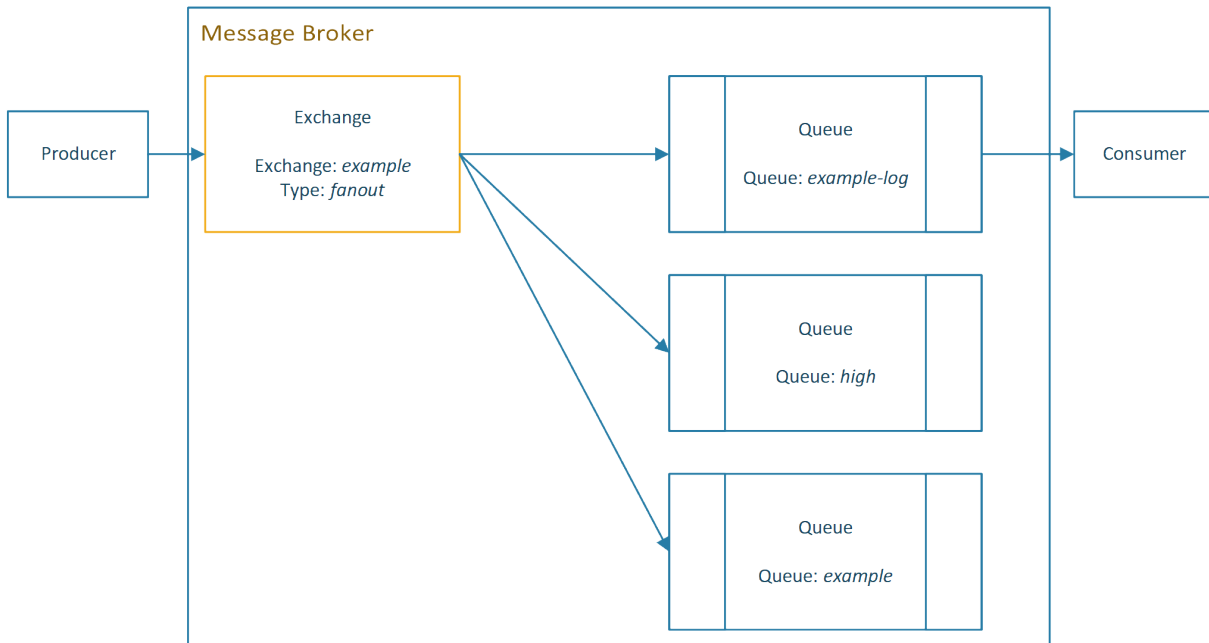


Figure 4.4: A fan-out exchange routing messages to three bound queues ignoring the routing key.

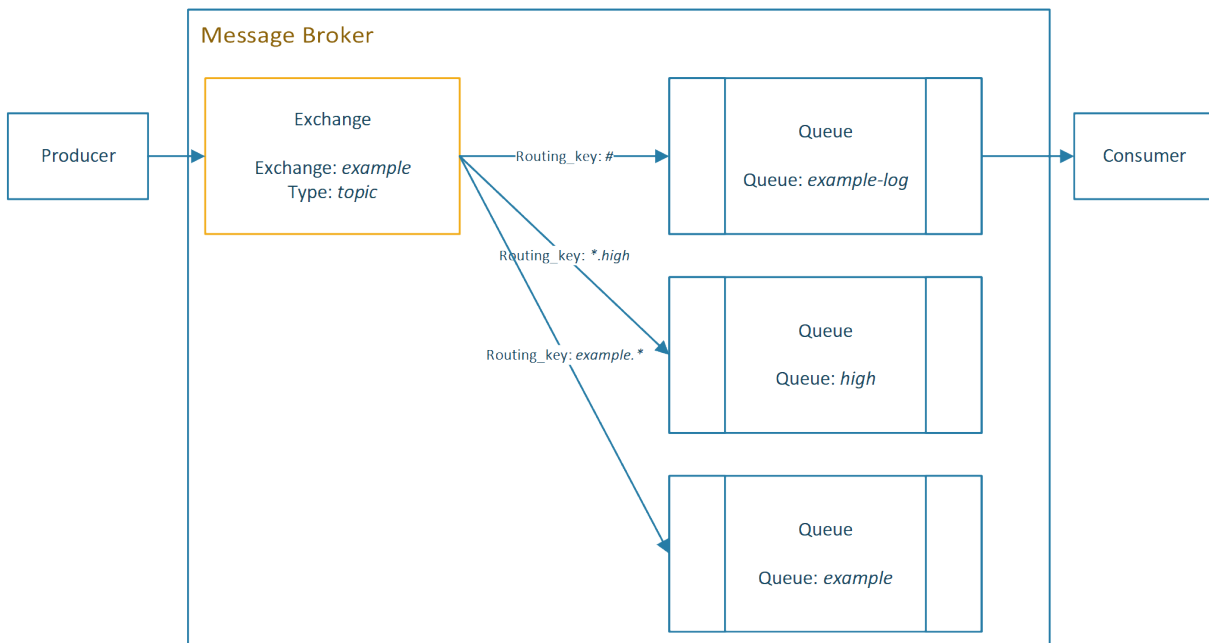


Figure 4.5: A topic exchange routing messages to three bound queues with three different routing key patterns.



- *\*.high* matches all routing keys ending with “*.high*”.
- *example.\** matches all routing keys starting with “*example.*”.

For example, if message *m* contains routing key “example.high” it would get delivered to all three message queues as it matches all three patterns.

#### 4.3.1.4 Header Exchange

A header exchange routes messages on multiple attributes in the message header instead of a routing key. It ignores the given routing key attribute and takes the attributes from the header attribute instead. If a queue is bound to a header exchange with more than one header to match, one more attribute is needed, namely, if all of the headers or just any must be matched. This is called the *x-match* argument. The benefit of a header exchange instead of a direct exchange is that the headers can be integers, hashes, etc., instead of a string.

```
queue_bind(
    exchange='example',
    queue='important-backend-logs', arguments={
        'severity': 'error',
        'host-group': 'backend',
        'x-match': 'all'})
```

Listing 4.3: Code snippet in python to bind queue *important-backend-logs* to header exchange *example* requires to match all header attributes. In particular, a message with *severity* matching *error* and *host-group* matching *backend* gets routed to the queue *important-backend-logs*.

#### 4.3.1.5 Combinations of Exchanges

The AMQP does not limit the user to binding an exchange directly to a queue. Exchanges can be bound together as well. For example, an application called *app* consists of background workers, loggers, and tracers. The background workers are responsible for cropping and resizing images. The tracers should receive all log messages, whereas the loggers should receive only logs with severity levels *critical* and *emergency*. A graphical representation of this application is in Figure 4.6. In Figure 4.6, all producers send their messages to the *Application* exchange, which is of the direct exchange type. The *Application* exchange sends all messages with a routing key starting with *image* to their bound queues. Messages with the routing key *logs* will be copied to the *traces* queue and to the *Log* exchange. Messages copied to the *Log* exchange will be processed further via this header exchange. Multiple consumers can get their messages from their desired queues. This enables the use of specialized applications or hardware for processing these messages.

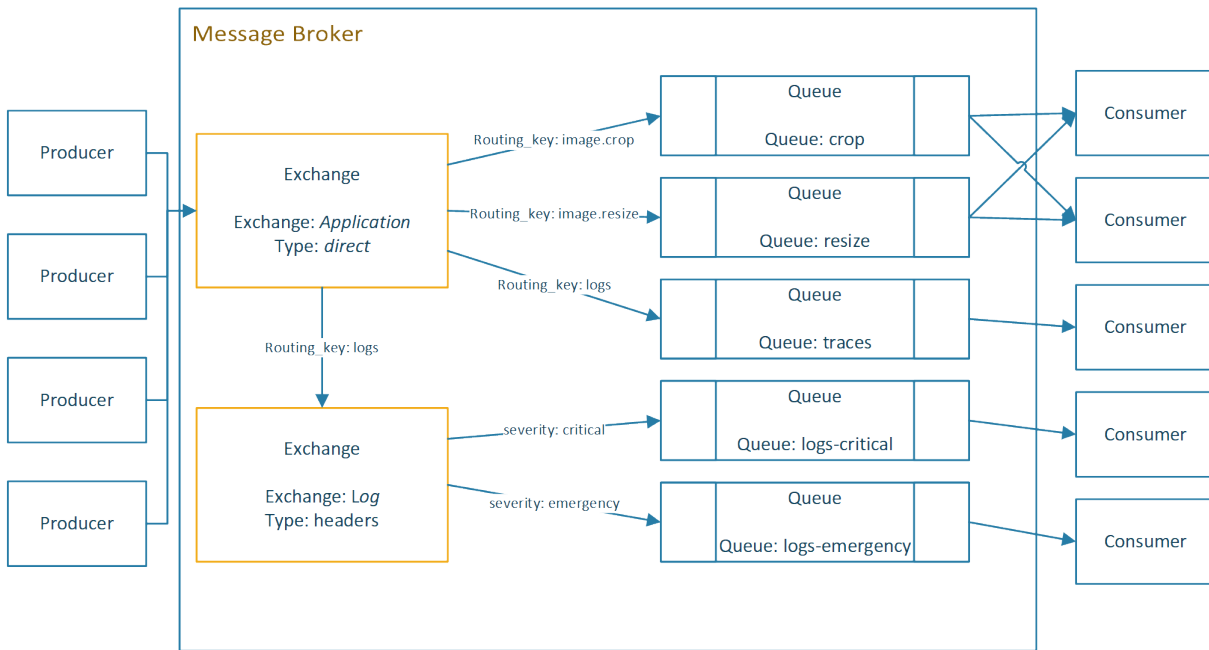


Figure 4.6: An example of an application with multiple exchanges and multiple queues.

#### 4.3.1.6 Message Acknowledgement

Sometimes it is necessary to acknowledge that a consumer successfully received or processed a message. As soon as the message broker receives the acknowledgement, the message is deleted from the queue. As networks and applications fail, the AMQP has the acknowledgement feature allotted. For example, a monitoring application may live with lost data and can acknowledge a message upon receiving, whereas a medical application may not. When a broker does not receive an expected acknowledgement, the message will be re-queued and delivered to other consumers, if connected to this queue [32].

### 4.3.2 RabbitMQ

One of the most popular and open-source message brokers is RabbitMQ [35]. RabbitMQ implements the AMQP, but is not limited to it; it supports others as well, such as the Hypertext Transfer Protocol (HTTP), Streaming Text Oriented Messaging Protocol (STOMP), and Message Queue Telemetry Transport (MQTT) protocols. RabbitMQ, despite the fact that many message brokers are centralized, can be deployed in a distributed manner to load-balance the workload, improve the throughput, and facilitate a federated deployment. Federation bears the benefit of connecting brokers together without being in the same network or same cluster, for example availability zones. RabbitMQ is extensible through plugins.

### 4.3.2.1 Scalability

Scalability is provided through plugins, such as *rabbitmq sharding* [8] and *rabbitmq consistent hash exchange* [7] but come with drawbacks. One caveat with sharding is, that the sharding plugin creates multiple queues behind the scenes to act as a big queue. The problem with this is that the numbers of consumers and shards for a queue need to be equal; otherwise, some queues will never get drained. Another drawback is, that the total ordering in the queue is lost. With the consistent hash exchange plugin the hashing is done on a routing key basis and not per message. The most basic way of implementing load balancing is to have the masters of queues on different nodes when RabbitMQ is deployed clustered (cf. section 4.3.2.2).

### 4.3.2.2 Availability

Availability is supported by clustering the RabbitMQ message broker. Clustering connects the nodes to a single logical broker. Exchanges are mirrored across all nodes, but queues are not by default. Queues are located on a single node unless otherwise requested. A client connecting to any node sees all queues in the cluster, though [33]. When a queue is mirrored, one *master* and several *mirrors* exist. All operations on a queue will be done on the master queue while being applied in the same order on the mirrors to keep the same state. If the master fails the longest running mirror is promoted to be the master [34].

## 4.3.3 ZeroMQ

ZeroMQ, also known as ØMQ, 0MQ, or zmq, is a radically different way of providing a message system. In contrast to the AMQP, where it is a family of messaging protocols and needs an implementation, for example, the ready-to-use application RabbitMQ, ZeroMQ is more like a build-your-own messaging system. ZeroMQ is a library of message functionality, which needs to be embedded in an application. Although the AMQP has the broker as a building block of messaging, ZeroMQ does not have a predefined broker, as it aims to be brokerless and therefore massively distributable. ZeroMQ does come with a library of broker-like functionality but needs to be included in every application where needed. Brokers are called *devices* in ZeroMQ parlance. ZeroMQ works like a Transmission Control Protocol (TCP) socket but is not limited to TCP, with added transport and messaging patterns. It treats queues as transport buffers and does not expose queues at all, unlike the AMQP, which requires that all queues be managed explicitly at one central place. Therefore, it appears “ [...] like having small brokers all over the network” [53]. These features allow to establish similar patterns and techniques as discussed in section 4.3, if implemented properly.

### 4.3.3.1 Availability and Scalability

Availability and scalability are not given a priori. The reason is that ZeroMQ is a library for building a custom-made distributed application for a particular problem. Therefore,

the availability and scalability depend on how this application is written with ZeroMQ.

# Chapter 5

## Evaluation

In the evaluation, the applications discussed in Chapter 4 are deployed singularly in different topologies, if suitable, as discussed in Chapter 3, or are combined to eliminate specific shortcomings. The evaluation files are stored on the attached CD (cf. Appendix B). Appendix A presents instructions for running discussed scenarios. The evaluation is a proof-of-concept, if the application can be used for designated topologies. Further, high availability and fault-tolerance is verified.

### 5.1 Evaluation Environment

The evaluation of the applications is done in virtual machines, running on a Microsoft Surface Pro 4 with an Intel i7-6650U CPU, 16GB RAM, 256GB SSD, and with Oracle's VirtualBox 5.1 as a hypervisor. Each VM is configured with two VCPUs, 1 GB RAM, 40 GB dynamic expanding virtual disk, and two network interfaces, one to communicate with the host and one to communicate directly between the VMs. At least two VMs get instantiated, whereas the last VM is always the client node.

#### 5.1.1 Vagrant

For reproducible development environments, a virtualization manager called Vagrant is used. Vagrant simplifies setup of multi-node clusters with an expected outcome. It can instantiate virtual machines with specified amount of memory or virtual central processing units (VCPUs), manages the network connectivity, and other PRs, which can be virtualized [10].

#### 5.1.2 Ansible

Ansible is a configuration automation tool. It configures systems and orchestrates environments with simplicity and ease-of-use. This enables software to be deployed on multiple

nodes and orchestrates them together [40]. Ansible is used to deploy the software used in the following sections.

## 5.2 Setting Up a MongoDB Server

With Vagrant, the environment of two nodes is set up quickly. Ansible provisions MongoDB on the first node in four steps. MongoDB recommends to install MongoDB from their own personal package archive (PPA) instead of the Ubuntu package repository, requiring Ubuntu to be instructed with adding the official MongoDB repository. After an update of the package list with `sudo apt-get update`, the installation of MongoDB is done with `sudo apt-get install mongodb-org`. Following the installation, `sudo systemctl enable mongod` ensures that MongoDB is automatically started upon a reboot. MongoDB is finally installed and ready to use.

The scenario files can be found on the CD at `/Demos/mongodb`.

### 5.2.1 Evaluation of a MongoDB Server

MongoDB as a database reflects the star topology. Therefore, it is important to ensure high availability and fault tolerance as discussed in Section 3.4. As expected, a single node neither provides both. Nevertheless, MongoDB supports a cluster deployment to fulfill this need, which is evaluated in Section 5.3.1.

One advantage of MongoDB is that OpenStack's telemetry service Ceilometer already uses it as a first class datastore [29]. If OpenStack is deployed with the telemetry service, MongoDB could be shared, reducing management work of an additional software.

## 5.3 Setting Up a MongoDB Cluster

Vagrant sets up a four nodes environment. Ansible provisions a MongoDB Cluster in a configuration shown in Figure 4.1. The *mongos* service is placed on the client node. Although, it can be placed separately on dedicated nodes [26]. The installation consists of five steps:

1. The *mongod* services are deployed.
2. The replication set between these *mongod* nodes is set up.
3. The configuration servers are deployed on the *mongod* nodes.
4. The *mongos* service is deployed on the client node.
5. The sharding information is given to the *mongos* service.

The deployment is done with Ansible files from the Ansible example repository [39] and adapted to a more recent version of Ansible. These files, expect CentOS 6 as an OS. Nevertheless, a MongoDB Cluster can be set up on Ubuntu, too.

The scenario files can be found on the CD at `/Demos/mongodb-cluster`.

### 5.3.1 Evaluation of a MongoDB Cluster

Requesting documents from a sharded cluster should contain the *shard key* [26]. If the shard key is included in the query, *mongos* can determine on which shards the requested documents are stored and only contact this subset of nodes. Otherwise, the query gets routed to any node, which represents a broadcast, which should be avoided. Thus, care must be taken to choose a shard key, upon which the most requests are made. Therefore, if the partial heaviness of users are stored (cf. Section 6 and Figure 6.2), the user ID is recommended to use as a shard key.

A synthetic data set has been used to evaluate fault tolerance and availability. The format of this data set is shown in Listing 5.1. The data set consists of 50 distinct tenant IDs, 30 distinct node IDs, and a random partial heaviness of users on the given node. For simplicity, each nodes hosts a VM of any user, resulting in a data set of 1500 documents. The data set can be found on the CD at `/Demos/mongodb-cluster/dataset.js`.

```
[
  {
    tenant_id: "GUID",
    node_id: "GUID",
    heaviness: "INTEGER"
  },{
    tenant_id: "GUID",
    node_id: "GUID",
    heaviness: "INTEGER"
  },
  .
  .
  .
]
```

Listing 5.1: Representation of the synthetic data set used to test sharding and replication.

When data is stored in this scheme, each node that wants to calculate the heaviness of an user requests documents with a matching *tenant\_id* key, and aggregates the returning documents. Every time a node updates its heaviness of users in MongoDB, it has to remove all documents belonging to itself. In case of a failed node, data of it would still be in the data store. MongoDB provides a feature to remove data after a specific time automatically, called *expireAfterSeconds* [19]. Thus, handling failures of the nodes transparently.

### 5.3.1.1 Availability and Fault Tolerance

The availability and fault tolerance can be tested with shutting down MongoDB nodes. Requesting a heaviness of an user, which was identified to be on shard *node1*, was still available after removing *node1*. The removal is done with `vagrant suspend node1`, which pauses the VM, rendering it unavailable for the other nodes. This proves, that MongoDB provides high availability and fault tolerance when using replica sets.

## 5.4 Setting Up a Redis Master-Slave Environment

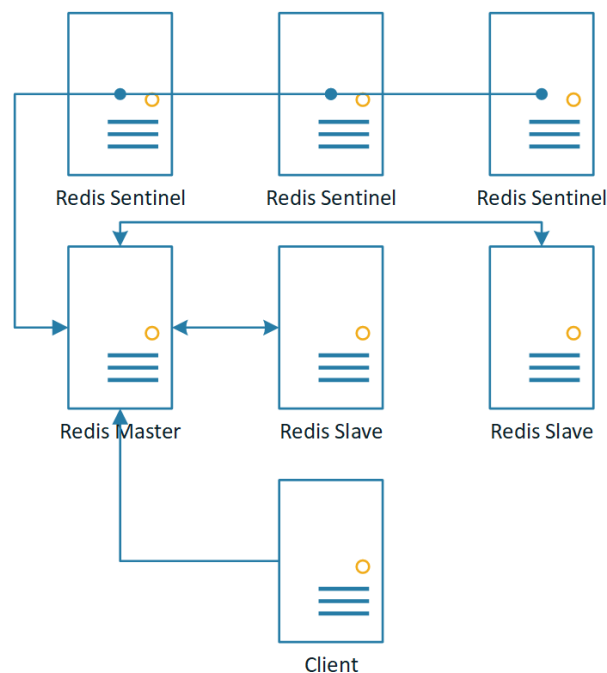


Figure 5.1: High available and fault tolerant Redis setup for evaluation.

Vagrant is configured to setup seven nodes, which Ansible provisions according Figure 5.1. The environment consists of one master and two slaves to provide redundancy. Three sentinels monitor the master, to provide a quorum of at least two and a failover of the master can take place. The provisioning is done with Ansible files from David Wittmann [51] and adapted for this environment.

The scenario files can be found on the CD at `/Demos/redis-cluster`.

### 5.4.1 Evaluation of a Redis Master-Slave Environment

Redis as a data store is a centralized node, thus it is designated for a star topology. Thereby, it must be made high available and fault tolerant. The evaluation is done with a synthetic data set and the Node.js client *node-redis-sentinel-client* from Ben Buckman



[4]. The data set consists of an array of key-value pairs containing 50 distinct tenant IDs and random user heaviness. The array contains 1500 key-value pairs.

In order that a node can calculate the heaviness of an user, it has to retrieve a list containing the heaviness of an user from each node accommodating a VM of that user. Thus, requires an expansible Redis data type, which the *list* fulfills.

#### 5.4.1.1 Availability and Fault Tolerance

After inserting the data set, a random key was picked. With the key, the list was retrieved from Redis. After this, the master node was suspended to let it failover to a slave. Subsequently, the same list with the same key was retrieved, showing that the failover has happened successfully and fault tolerance is given.

#### 5.4.1.2 Redis Expiration

In Redis, expiration dates can only be set on keys and not values [41]. Thus, makes it difficult to remove stale data automatically from complex data structures, such as a list.

## 5.5 RabbitMQ

RabbitMQ provides a simple approach to be high available, with the caveat of lesser throughput. Also, RabbitMQ has limited horizontal scalability, Although, it scales vertically. Thus, it was forbore from doing an implementation. Nevertheless, some implementation options are discussed.

Using RabbitMQ in a star topology as a central node as a broadcaster is not feasible. As RabbitMQ is not a data store, only self-contained messages can be distributed with it. Thus, a ring topology is desired, to get a linear message volume. For example, for each participating node, a queue with the index of the node within the ring as a name should be created, denoted as the *output queue*. Thus, the message passed in these queues, can be recovered by checking its output queue if the message is still there. After the recovery, the orphaned queue could get removed. Still, a master node is required to maintain the order. Although, it would be more practical to implement it in ZeroMQ, which is more flexible and a custom application has to be programmed anyway.

## 5.6 ZeroMQ

As discussed in Section 4.3.3, ZeroMQ is not an application per se, but an embeddable messaging library. Thus, it cannot be evaluated without being implemented in an application. Although, some implementation options are given.

ZeroMQ could be implemented in a partially meshed topology. For example, a node only subscribes to notifications of users it hosts VMs for, as it is assumed that not all nodes provides hosting for the same set of users, results in a partially meshed topology. As ZeroMQ is inherently distributed, it needs to know where to subscribe from, in particular which nodes hosts VMs of which user. OpenStack’s ZeroMQ implementation uses a Redis server as a “matchmaker”, which is described in [16]. A similar approach could be used in this topology. Each node stores its endpoint at the Redis server with the users as the key, for example, in a list structure (cf. Figure 5.2). Thus, a node looks its users up and subscribes the endpoints from the retrieved nodes. The lookup table is only needed during build up of the connections. Nevertheless, if there is a change in topology, or a new user appears on a node, all nodes have to recheck the lookup table and do adjustments of the subscriptions. Therefore, in a busy cloud this implementation may not be wanted. Another caveat in Redis, is the expiration only applies on keys and not on values (cf. Section 5.4.1.2). Thus, an algorithm to remove lost nodes from the Redis lists must be defined.

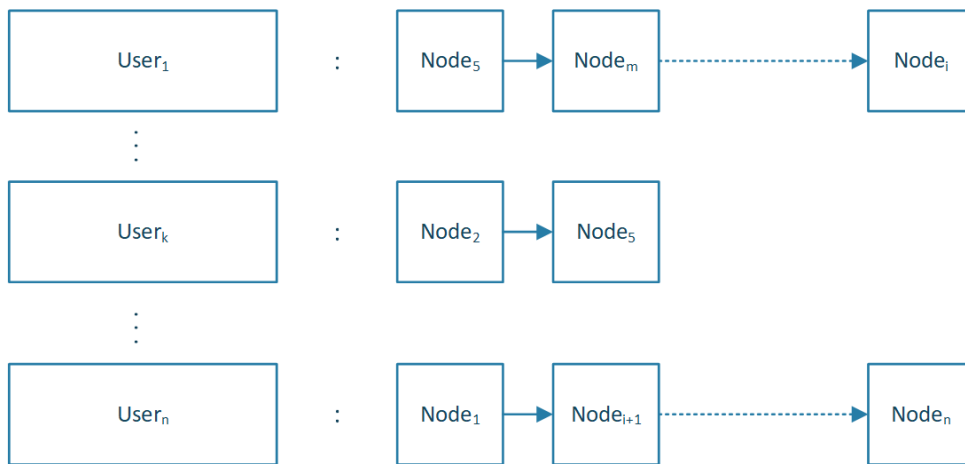


Figure 5.2: Lookup table to find nodes providing hosting for specific users.

Another implementation option is to use ZeroMQ in a ring topology. A similar approach as the aforementioned can be made. Storing its endpoint in a Redis list, the order in a ring could be established. Redis can notify nodes via its publish-subscribe pattern about changes on a key [44]. Currently, the notification is not reliable, that means, when a client disconnects and reconnects for a period of time, all notifications sent during that time, will be lost. Thus, this renders it unfeasible. Another solution would be, as a node only connects to its successor, to establish a heartbeat to detect a failure, or simple, when the vector cannot be sent to the successor. When a failure occurs, a simple solution would be, to look up the successor of the node which failed and send the vector to this node.

When the API discussed in Chapter 6 is implemented, several implementations with ZeroMQ can be conveniently tested.

## 5.7 Combinations

The implementation options do not have to be standalone. Combinations allow to mitigate drawbacks of single topologies or applications.

### 5.7.1 MongoDB and Redis

Redis is often used as a cache [17] layer between the data store layer and the application layer. The idea behind this setup is to reduce the workload on the data store layer. As there are many same queries against the data store layer from different nodes, e.g. get heaviness of user  $u_k$ , but the result does not change for  $\mu$  seconds the result of the query could be cached in the Redis server. This reduces the calculation load on the data store layer. The trade-off would be an increased message count, as in the worst case the first step (i) the cache layer must be contacted if it has a precalculated value stored. If it does, it is called a cache-hit, if it does not, it is called a cache-miss. If not (ii), one of the compute nodes reach out to the data store layer and get the new heaviness of users from it, calculate the new heaviness of users, which then (iii) the node stores in the cache layer for other nodes. Thus, the expiration of keys in Redis can be used.

Another option for this implementation is to use the publish-subscribe mode of Redis. This way, the subsequent accesses at the cache layer will save one trip, as the result would be distributed to the nodes, which are interested in that particular result.



# Chapter 6

## Designing of an API

To propose an API, first, the parts required to calculate the heaviness of users were identified. These were NRIs, RUIs, CRS,  $s'$ ,  $H_V$ s, and partial  $H_U$ s.

These parts were mapped to the entity, which had knowledge of these. In Figure 6.1, it is shown that the API has knowledge about or knows how to get information about the CRS,  $s'$ , and partial  $H_U$ s, for example. Meanwhile, the FS is can calculate the  $H_V$  set of its VMs with the ability to get its RUIs and the heaviness of users when it receives partial  $H_U$ s. The FS also knows about its nodes' NRIs and delivers them to the API.

The procedure for the FS to communicate with the API is shown in Figure 6.2 in a sequence diagram.

This sequence in Figure 6.2 enables the possible messaging implementations to act as a server responding to a client, but this order must be maintained. When this order is kept, the sequence in the FS to calculate the heaviness of an user does not need to be adapted to different messaging implementations. An example for three topologies and their operation executions is given in Table 6.1.

For operation *request of all*  $H_U$ , the old partial  $H_U$  must be in the return vector as well. The reason for this is that the FS always subtracts its old partial  $H_U$  upon aggregation for the full  $H_U$ . This, enables messaging patterns that keep the state of the old  $H_U$  in the full  $H_U$  and must be subtracted, such as the ring topology.

The proposed API consists of six methods, namely:

**Update NRI** This method is necessary for updating the CRS.

**Request CRS** This method is responsible for calculating the CRS or for retrieving NRIs as needed. Finally, the CRS must be returned.

**Request  $s'$  ( $aq(u_i)$ )** This method retrieves the value from a responsible entity.

**Update partial  $H_U$**  The implementation of this method may differ from one topology to another, but is mainly responsible for being sent somewhere where this value is needed.

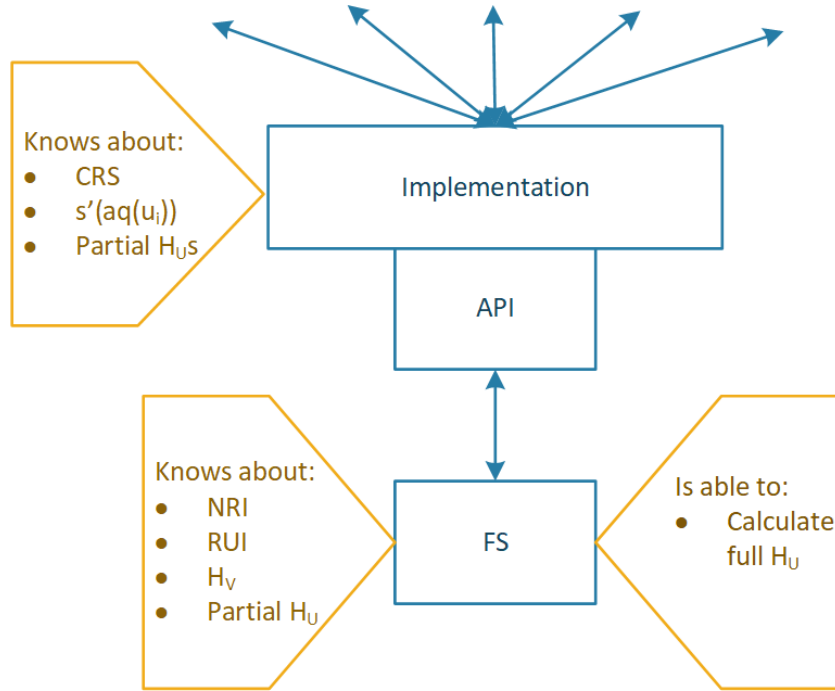


Figure 6.1: Knowledge of parts required for heaviness of users calculation.

Operation	Full Mesh	Ring	Star
Receive NRI	Send NRI to the remaining $n - 1$ nodes.	Wait until vector arrives add NRI to vector. Send vector to successor.	Send NRI to central node.
Request of CRS and $s'(aq(u_i))$	Wait for all $n - 1$ nodes' NRI and respond with CRS. Acquire $s'(aq(u_i))$ from responsible entity and respond with it.	Wait until vector arrives read CRS from vector and respond with it. Acquire $s'(aq(u_i))$ from responsible entity and respond with it.	Request CRS from central node and respond with it. Acquire $s'(aq(u_i))$ from responsible entity and respond with it.
Receives partial $H_U$	Send partial $H_U$ to all remaining nodes $n - 1$ .	Wait until vector arrives attach partial $H_U$ to received vector.	Send partial $H_U$ to central node.
Request of all $H_U$	Wait for all $n - 1$ nodes' $H_U$ , attach old and new partial $H_U$ to the set, and respond with it.	Respond with vector.	Request all $H_U$ from central node, attach old partial $H_U$ to it, and respond with it.
receive full $H_U$	Do nothing.	Update vector and send it to successor.	Do nothing.

Table 6.1: Example of operations that the messaging implementations complete with different topologies behind the API upon request of the FS.

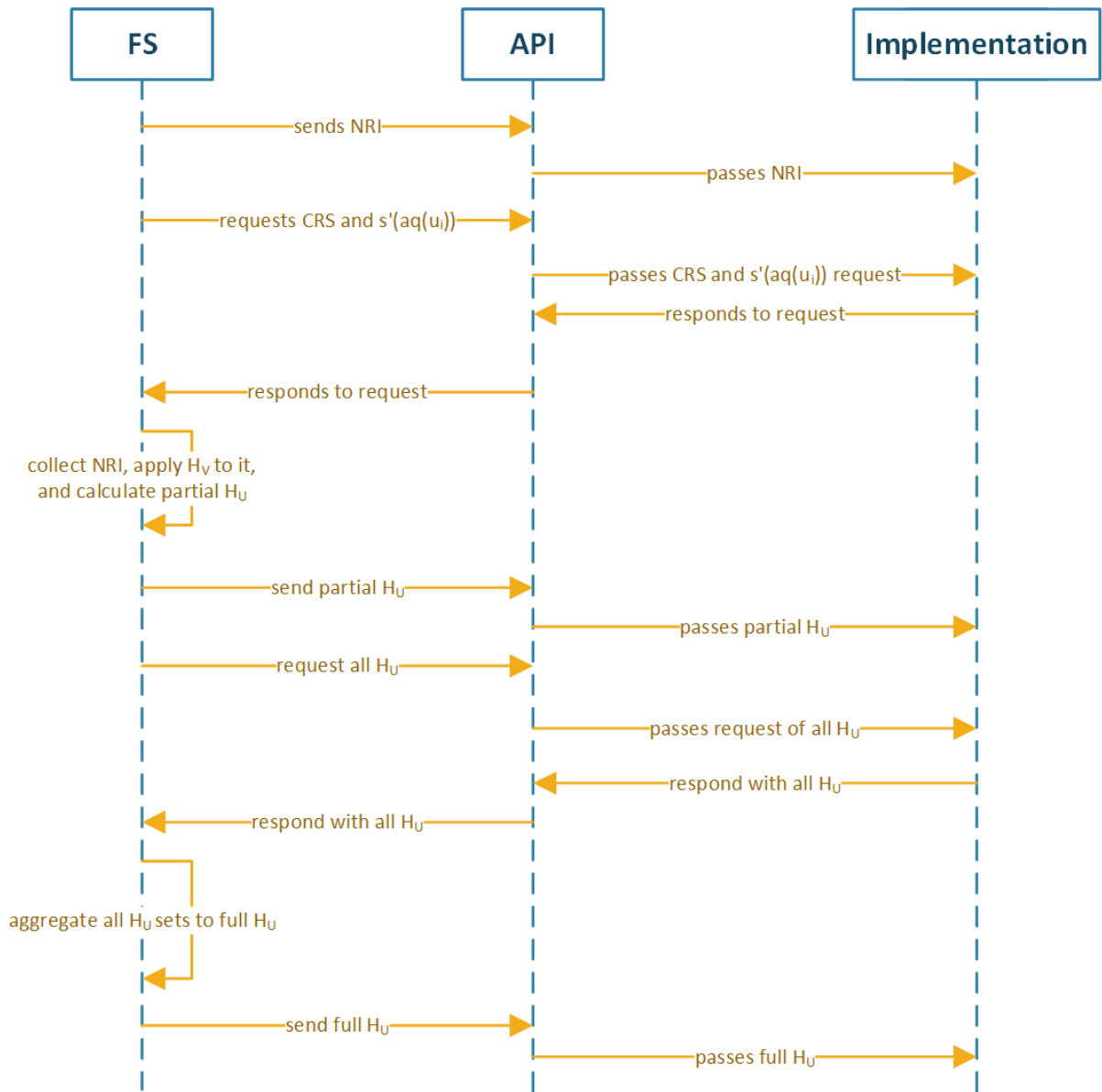


Figure 6.2: Interaction between FS and API for a calculation of heaviness of users.

**Request all  $H_U$**  This method is responsible for retrieving all  $H_U$ s, including the old and new partial  $H_U$  of the requesting FS.

**Update  $H_U$**  The implementation of this method may differ from one topology to another, but can be used to inform other nodes about the new  $H_U$ .



# Chapter 7

## Summary and Conclusions

The development of logical message flow topologies in Chapter 3 has shown that a reduction of a linear message volume is possible, namely the ring and the star topology. Also, the pitfalls and caveats of an implementation have been highlighted. The ring topology may be the best in terms of the message volume, but is difficult to implement efficiently, such as the master election, the master failover, and recovery from a lost node or vector. Thus, the implementation has to be done carefully, or the system will break easily. The star topology is easier to implement, but care must be taken to make the central node highly available and fault tolerant.

Taking into account that MongoDB is easily made scalable and highly available, featuring sharding, built-in replication, and a query router, the recommended implementation is a star topology.

The API presented in Chapter 6 allows for the flexible implementation of different topologies. An implementation must expose the six proposed methods to the FS, and result in no change in the sequence of the FS program.

### 7.1 Future Work

The ring topology needs a rock-solid implementation, which could be achieved with ZeroMQ. Still, solutions for the mentioned implementation difficulties must be found.

As message volume reduction is a key point, the developed logical message flow topologies should be implemented with the presented API and then benchmarked to test their efficiency.

The topology, bandwidth, and transport of a network could be evaluated, because, for example, a single message can get splitted into multiple network packets.



# Bibliography

- [1] Aerospike, Inc. *What is a Key-Value Store?* [Online; accessed June 5th, 2017]. URL: <http://www.aerospike.com/what-is-a-key-value-store/>.
- [2] B. Albert and A. P. Jayasumana. *FDDI and FDDI-II: architecture, protocols, and performance*. first. Artech House Telecommunications Library. Artech House, Jan. 1994.
- [3] A. B. Bondi. “Characteristics of Scalability and Their Impact on Performance”. In: *Proceedings of the 2Nd International Workshop on Software and Performance*. WOSP '00. Ottawa, Ontario, Canada: ACM, 2000, pp. 195–203. ISBN: 1-58113-195-X. DOI: 10.1145/350391.350432. URL: <http://doi.acm.org/10.1145/350391.350432>.
- [4] B. Buckmann. *Redis Sentinel Client for Node.js*. [Online; accessed June 8th, 2017]. URL: <https://github.com/DocuSignDev/node-redis-sentinel-client>.
- [5] CoreOS, Inc. *Getting started with etcd*. [Online; accessed June 5th, 2017]. URL: <https://coreos.com/etcd/docs/latest/getting-started-with-etcd.html>.
- [6] S. B. Davidson, H. Garcia-Molina, and D. Skeen. “Consistency in a Partitioned Network: A Survey”. In: *ACM Comput. Surv.* 17.3 (Sept. 1985), pp. 341–370. ISSN: 0360-0300. DOI: 10.1145/5505.5508. URL: <http://doi.acm.org/10.1145/5505.5508>.
- [7] GitHub. *RabbitMQ Consistent Hash Exchange Type*. [Online; accessed June 9th, 2017]. URL: <https://git.io/vHAH1>.
- [8] GitHub. *RabbitMQ Sharding Plugin*. [Online; accessed June 9th, 2017]. URL: <https://git.io/vHAHD>.
- [9] J. Hammons. *Bash on Ubuntu on Windows - Installation Guide*. [Online; accessed May 2nd, 2017]. 2017. URL: [https://msdn.microsoft.com/en-us/commandline/wsl/install\\_guide](https://msdn.microsoft.com/en-us/commandline/wsl/install_guide).
- [10] Hashicorp. *Introduction to Vagrant*. [Online; accessed May 1st, 2017]. URL: <https://www.vagrantup.com/intro/index.html>.
- [11] Hashicorp. *Vagrant and Windows Subsystem for Linux*. [Online; accessed May 2nd, 2017]. URL: <https://www.vagrantup.com/docs/other/wsl.html>.
- [12] M. Helsley. *LXC: Linux container tools*. [Online; accessed June 5th, 2017]. 2009. URL: <https://www.ibm.com/developerworks/linux/library/l-lxc-containers/>.

- [13] M. D. Hill. “What is Scalability?” In: *SIGARCH Comput. Archit. News* 18.4 (Dec. 1990), pp. 18–21. ISSN: 0163-5964. DOI: 10.1145/121973.121975. URL: <http://doi.acm.org/10.1145/121973.121975>.
- [14] john@rjohara.com. *AMQP 1.0 Becomes OASIS Standard*. [Online; accessed June 7th, 2017]. URL: <http://www.amqp.org/node/102>.
- [15] M. Gentz, S. Deng, lucasfmo, and C. Caserio. *NoSQL im Vergleich zu SQL*. [Online; accessed April 16th, 2017]. Mar. 14, 2017. URL: <https://docs.microsoft.com/de-de/azure/documentdb/documentdb-nosql-vs-sql>.
- [16] S. Mannhart. “Development and Evaluation of an OpenStack Extension to Enforce Cloud-wide, Multi-resource Fairness during VM Runtime”. bachelor thesis. Zürich, Switzerland: Universität Zürich, Communication Systems Group, Department of Informatics, Mar. 2016.
- [17] Microsoft Corporation. *Azure Redis Cache*. [Online; accessed June 5th, 2017]. URL: <https://azure.microsoft.com/de-de/services/cache/>.
- [18] Microsoft Corporation. *Was ist ein Azure AD-Verzeichnis?* [Online; accessed May 31st, 2017]. URL: [https://msdn.microsoft.com/library/azure/jj573650.aspx#BKMK\\_WhatIsAnAzureADTenant](https://msdn.microsoft.com/library/azure/jj573650.aspx#BKMK_WhatIsAnAzureADTenant).
- [19] MongoDB, Inc. *Expire Data from Collections by Setting TTL*. [Online; accessed April 30th, 2017]. URL: <https://docs.mongodb.com/manual/tutorial/expire-data/>.
- [20] MongoDB, Inc. *mongo*. [Online; accessed May 26th, 2017]. URL: <https://docs.mongodb.com/manual/reference/program/mongo/>.
- [21] MongoDB, Inc. *mongo*. [Online; accessed May 26th, 2017]. URL: <https://docs.mongodb.com/manual/core/replica-set-members/>.
- [22] MongoDB, Inc. *mongod*. [Online; accessed May 26th, 2017]. URL: <https://docs.mongodb.com/manual/reference/program/mongod/>.
- [23] MongoDB, Inc. *MongoDB Architecture*. [Online; accessed April 12th, 2017]. URL: <https://www.mongodb.com/mongodb-architecture>.
- [24] MongoDB, Inc. *MongoDB Package Components*. [Online; accessed May 26th, 2017]. URL: <https://docs.mongodb.com/manual/reference/program/>.
- [25] MongoDB, Inc. *mongos*. [Online; accessed May 26th, 2017]. URL: <https://docs.mongodb.com/manual/reference/program/mongos/>.
- [26] MongoDB, Inc. *mongos*. [Online; accessed June 7th, 2017]. URL: <https://docs.mongodb.com/manual/core/sharded-cluster-query-router/>.
- [27] MongoDB, Inc. *Storage Engines*. [Online; accessed April 30th, 2017]. URL: <https://docs.mongodb.com/manual/core/storage-engines/>.
- [28] MongoDB, Inc. *What is MongoDB?* [Online; accessed April 12th, 2017]. URL: <https://www.mongodb.com/what-is-mongodb>.
- [29] OpenStack Foundation. *Configuring the Telemetry (ceilometer) service*. [Online; accessed May 1st, 2017]. URL: <https://docs.openstack.org/developer/openstack-ansible/mitaka/install-guide/configure-ceilometer.html>.

- [30] OpenStack Foundation. *OpenStack Wiki: ZeroMQ*. [Online; accessed March 10th, 2017]. 2014. URL: <https://wiki.openstack.org/wiki/ZeroMQ>.
- [31] OpenStack Foundation. *Tenant*. [Online; accessed May 31st, 2017]. URL: <https://wiki.openstack.org/wiki/Tenant>.
- [32] Pivotal Software, Inc. *AMQP 0-9-1 Model Explained*. [Online; accessed June 7th, 2017]. URL: <https://www.rabbitmq.com/tutorials/amqp-concepts.html>.
- [33] Pivotal Software, Inc. *Clustering Guide*. [Online; accessed June 9th, 2017]. URL: <https://www.rabbitmq.com/clustering.html>.
- [34] Pivotal Software, Inc. *Highly Available (Mirrored) Queues*. [Online; accessed June 9th, 2017]. URL: <https://www.rabbitmq.com/ha.html>.
- [35] Pivotal Software, Inc. *RabbitMQ - Messaging that just works*. [Online; accessed June 8th, 2017]. URL: <https://www.rabbitmq.com/#getstarted>.
- [36] Pivotal Software, Inc. *Understanding NoSQL*. [Online; accessed April 16th, 2017]. URL: <https://spring.io/understanding/NoSQL>.
- [37] P. Poullie, S. Mannhart, and B. Stiller. *Defining and Enforcing Fairness Among Cloud Users by Adapting Virtual Machine Priorities During Runtime*. Tech. rep. IFI-2016.04. Zürich, Switzerland: Universität Zürich, Mar. 2016. URL: <https://files.ifi.uzh.ch/CSG/staff/poullie/extern/publications/IFI-2016.04.pdf>.
- [38] P. Poullie, S. Mannhart, and B. Stiller. “Virtual machine priority adaption to enforce fairness among cloud users”. In: *2016 12th International Conference on Network and Service Management (CNSM)*. Oct. 2016, pp. 91–99. DOI: 10.1109/CNSM.2016.7818404.
- [39] Red Hat, Inc. *GitHub - ansible/ansible-examples*. [Online; accessed May 22nd, 2017]. URL: <https://git.io/vHAHi>.
- [40] Red Hat, Inc. *HOW ANSIBLE WORKS*. [Online; accessed May 30th, 2017]. URL: <https://www.ansible.com/how-ansible-works>.
- [41] Redis Labs. *EXPIRE key seconds*. [Online; accessed June 5th, 2017]. URL: <https://redis.io/commands/expire>.
- [42] Redis Labs. *Introduction to Redis*. [Online; accessed June 5th, 2017]. URL: <https://redis.io/topics/introduction>.
- [43] Redis Labs. *Redis cluster tutorial*. [Online; accessed June 5th, 2017]. URL: <https://redis.io/topics/cluster-tutorial>.
- [44] Redis Labs. *Redis Keyspace Notifications*. [Online; accessed June 5th, 2017]. URL: <https://redis.io/topics/notifications>.
- [45] Redis Labs. *Redis Sentinel Documentation*. [Online; accessed June 5th, 2017]. URL: <https://redis.io/topics/sentinel>.
- [46] M. Rouse. *What is high availability (HA)? - Definition from WhatIs.com*. [Online; accessed May 21st, 2017]. Sept. 2005. URL: <http://searchdatacenter.techtarget.com/definition/high-availability>.
- [47] M. Rouse. *What is scalability? - Definition from WhatIs.com*. [Online; accessed May 21st, 2017]. Apr. 2006. URL: <http://searchdatacenter.techtarget.com/definition/scalability>.

- [48] L. Schubert and K. Jeffery. *Advances in clouds: Research in future cloud computing*. Tech. rep. Luxembourg: European Commission, Publications Office of the European Union, 2012.
- [49] Teach-ICT.com, Ltd. *Star Networks*. [Online; accessed May 17th, 2017]. URL: [http://www.teach-ict.com/gcse\\_new/networks/topologies/miniweb/pg4.htm](http://www.teach-ict.com/gcse_new/networks/topologies/miniweb/pg4.htm).
- [50] Wikipedia. *Database normalization* — *Wikipedia, The Free Encyclopedia*. [Online; accessed May 30th, 2017]. 2017. URL: [https://en.wikipedia.org/w/index.php?title=Database\\_normalization&oldid=781823261](https://en.wikipedia.org/w/index.php?title=Database_normalization&oldid=781823261).
- [51] D. Wittmann. *ansible-redis*. [Online; accessed June 8th, 2017]. URL: <https://github.com/DavidWittman/ansible-redis#master-slave-replication>.
- [52] WSO2, Inc. *Key Concepts*. [Online; accessed June 7th, 2017]. URL: <https://docs.wso2.com/display/EI611/Key+Concepts#KeyConcepts-Messagetransformation>.
- [53] ZeroMQ Community. *Welcome from AMQP*. [Online; accessed May 30th, 2017]. URL: <http://zeromq.org/docs:welcome-from-amqp>.

# Abbreviations

<b>AMQP</b>	Advanced Message Queuing Protocol
<b>BLOB</b>	Binary Large Object
<b>BSON</b>	Binary JavaScript Object Notation
<b>CRC</b>	cyclic redundancy check
<b>CRS</b>	cloud resource supply
<b>CSG</b>	Communications Systems Group
<b>FDDI</b>	Fiber Distributed Data Interface
<b>FS</b>	Fairness Service
<b>GM</b>	Greediness Metric
<b>HTTP</b>	Hypertext Transfer Protocol
<b>JSON</b>	JavaScript Object Notation
<b>MQTT</b>	Message Queue Telemetry Transport
<b>NoSQL</b>	not only SQL
<b>NRI</b>	node resource information
<b>OASIS</b>	Organization for the Advancement of Structured Information Standards
<b>OS</b>	operating system
<b>PPA</b>	personal package archive
<b>PR</b>	physical resource
<b>RTT</b>	round trip time
<b>RUI</b>	runtime utilization information
<b>SQL</b>	structured query language
<b>STOMP</b>	Streaming Text Oriented Messaging Protocol
<b>TCP</b>	Transmission Control Protocol

<b>TTL</b>	time to live
<b>VCPU</b>	virtual central processing unit
<b>VM</b>	virtual machine
<b>VR</b>	virtual resource



# List of Figures

3.1	Representation of a fully meshed topology. . . . .	6
3.2	Representation of a ring topology. . . . .	6
3.3	Representation of a star topology. . . . .	7
3.4	Adding a new node to the fully meshed topology. . . . .	8
3.5	Adding a new node to the ring topology. . . . .	9
3.6	Representation of a multi-ring topology. . . . .	10
3.7	Removing a node in a ring topology in two different cases. . . . .	10
4.1	Architecture of a three node sharded MongoDB cluster with replica sets and the retrieval process for high availability, fault tolerance, and horizontal scalability. . . . .	22
4.2	Concept of message acquisition, routing, queuing, and delivering in the AMQP. . . . .	27
4.3	A direct exchange called <i>example</i> delivers messages with routing key <i>K example</i> to the bound queue called <i>example-jobs</i> . . . . .	27
4.4	A fan-out exchange routing messages to three bound queues ignoring the routing key. . . . .	28
4.5	A topic exchange routing messages to three bound queues with three different routing key patterns. . . . .	28
4.6	An example of an application with multiple exchanges and multiple queues. . . . .	30
5.1	High available and fault tolerant Redis setup for evaluation. . . . .	36
5.2	Lookup table to find nodes providing hosting for specific users. . . . .	38
6.1	Knowledge of parts required for heaviness of users calculation. . . . .	42
6.2	Interaction between FS and API for a calculation of heaviness of users. . . . .	43



# List of Tables

6.1	Example of operations that the messaging implementations complete with different topologies behind the API upon request of the FS. . . . .	42
-----	--	----



# Appendix A

## Installation Guidelines

The installation instructions are different for Microsoft Windows 10, hereinafter referred to as “Windows”, than for Canonical Ubuntu 16.04, hereinafter referred to as “Ubuntu”. Although the demos run on Windows, it is recommended to use Ubuntu.

In a terminal, special indicators are used as follows:

- \$ denotes the standard user
- # denotes the root user

### A.1 Windows Preparations

#### A.1.1 1. Step - Installing Oracle’s VirtualBox

The installation package for Oracle’s VirtualBox 5.1 is located on the CD in the folder `\Software\Windows\VirtualBox-5.1.22-115126-Win.exe` or can be downloaded from the product’s website <https://www.virtualbox.org/wiki/Downloads>.

#### A.1.2 2. Step - Installing Windows Subsystem for Linux (WSL)

Ansible does not support Windows as a control machine. Although, Windows 10 supports a subsystem to support Linux to a certain degree, which can be used to run Ansible. The requirement to use WSL is to have at least Windows 10 Anniversary Update (build 14393) or later. The instructions are taken from [9] and a copy of this website is located on the CD at `Websites\InstallationInstructions\BashonUbuntuonWindows-InstallationGuide.html`. The installation is done in two steps:

1. Turn-on developer mode

- (a) Open Settings → Update and Security → For developers
  - (b) Select the Developer Mode radio button
2. Enable the “Windows Subsystem for Linux (beta)” feature via the command-line:
    - (a) Open a PowerShell prompt as administrator and run:
 

```
Enable-WindowsOptionalFeature \
-Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

After the installation of the WSL, a restart is required. After the restart, a command prompt can be opened and the *bash* command can be run.

### A.1.3 3. Step - Installing Vagrant

The installation of Vagrant in WSL is the same as on Ubuntu (cf. Section A.2.2). After installing Vagrant in WSL, there are some caveats using WSL. WSL support from Vagrant is considered alpha. WSL puts a layer of isolation on to the Windows system, but Vagrant needs access to programs installed on Windows and not WSL, e.g., *VBoxManage.exe* from VirtualBox. Vagrant can be guided to use these files directly from the Windows system instead of WSL. For this, an environment variable has to be set:

```
export VAGRANT_WSL_ENABLE_WINDOWS_ACCESS="1"
```

Furthermore, in Windows the executables used by Vagrant need to be in the environment variable *PATH*. For Vagrant it is essential to have access to the *VBoxManage.exe*. Therefore, the installation path of this executable must be added to the *PATH* environment variable, e.g., `C:\ProgramFiles\Oracle\VirtualBox`. More information about this caveat can be found at [11].

### A.1.4 4. Step - Move files from Windows to WSL

To be able to use the demos from the CD in the WSL, the files have to be moved to the WSL.

1. The folder `\SourceCode\Demos` need to be copied from the CD to, e.g., `C:\`
2. In a WSL bash terminal following commands need to be entered:

```
# mv /mnt/c/Demos ~/
```

The demos are now available at the path `~/Demos` in WSL. The following steps are the same as for Ubuntu, beginning from section A.2.3.

## A.2 Ubuntu Preparations

### A.2.1 1. Step - Installing Oracle's VirtualBox

```
$ wget \
-q https://www.virtualbox.org/download/oracle_vbox_2016.asc \
-O- | sudo apt-key add -
$ wget -q https://www.virtualbox.org/download/oracle_vbox.asc \
-O- | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install virtualbox-5.1
```

### A.2.2 2. Step - Installing Vagrant

The Vagrant package in the Ubuntu repository is outdated. A more recent version can be downloaded from the Vagrant website as a Debian package or from the CD located at `/Software/Ubuntu/vagrant_1.9.5_x86_64.deb`:

```
$ sudo dpkg -i vagrant_1.9.5_x86_64.deb
$ vagrant -v
```

The output should be:

```
Vagrant 1.9.5
```

### A.2.3 3. Step - Installing Ansible

Ansible is installed from the terminal as follows:

```
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

### A.2.4 4. Step - Running Demos

The demos, in particular scenario files, are located on the CD at `/Demos`. Thus, the current working directory has to be changed to the CD path.

The demos contain a succinct `README` file, to point out what this scenario does and potential special instructions. The general approach to run a scenario is to change into the folder of the desired scenario, e.g., for the `mongodb-cluster` `$ cd Demos/mongodb-cluster`. Thereafter, `$ vagrant up` starts the provisioning of the scenario and `$ vagrant destroy` tears the environment down.





# Appendix B

## Contents of the CD

The contents of the CD is as follows:

**Abstract** is an unformatted abstract in English.

**BA.zip** contains this thesis as  $\text{\LaTeX}$  source file.

**Bachelorarbeit.pdf** contains this thesis in a PDF.

**Bachelorarbeit.ps** contains this thesis in PS.

**Content** contains this listing as digital reference.

**Demos** contains scenario files for the evaluation.

**Software** contains software, which is needed to run the scenario files.

**Visios** contains Microsoft Visio files and images used in this thesis.

**Websites** contains websites, of which citations were made.

**Zusfsg** is an unformatted abstract in German.