

BACHELOR THESIS

Extension of a SCADA Framework to support High Availability and Authenticated Encryption

#Ruby #ØMQ #NaCl

Patrik Wenger, Manuel Schuler

client: mindclue GmbH

supervisor: Prof. Dr. Farhad Mehta

expert: Prof. John Doe

September – December, 2016

Abstract

Declaration of Originality

We hereby confirm that we are the sole authors of this document, the described changes to the Roadster framework, and libraries developed as a byproduct. Unless stated differently, all illustrations in this document are our creations.

Acknowledgements

Special thanks to Pieter Hintjens † (3 December 1962 – 4 October 2016) for his amazing work and contagious passion within the ØMQ and distributed computing communities. We send our deepest condolences to his family. Rest in peace.

Contents

I	Management Summary	1
1	Context	2
1.1	Initial Situation	2
1.2	Goals	2
1.3	Software development process	2
1.4	Project management infrastructure	3
2	Project Phases	4
2.1	Plan	4
2.2	Inception	4
2.3	Elaboration	4
2.4	Construction	4
2.5	Transition	4
3	Results	5
II	Technical Report	6
4	Scope	7
4.1	Motivation	7
4.1.1	Open-Source engagement	8
4.2	Initial Situation	8
4.2.1	mindclue GmbH	8
4.2.2	Roadster	9
4.2.3	ØMQ	9
4.2.4	Software architecture	10
4.3	Goals	13
5	Requirements	14
5.1	Functional	14
5.1.1	Cluster	14
5.1.2	High availability	16
5.1.3	Persistence synchronization	18
5.1.4	OPC UA HA	19
5.2	Use Cases	19
5.2.1	UC01: Cluster	19
5.2.2	UC02: Hardware failure at top	19
5.2.3	UC03: Hardware failure at bottom	19
5.2.4	UC04: Persistence synchronization	19
5.2.5	UC05: OPC-UA HA	19
5.3	Non-Functional Requirements	20
5.3.1	Simplicity	20
5.3.2	Testing	20

5.3.3	Security	20
5.3.4	Coding Guidelines	20
6	Approach	21
6.1	Getting familiar with Roadster	21
6.2	Testing	21
6.2.1	Setup	21
6.2.2	Unit tests	22
6.2.3	Integration tests	22
6.2.4	Continuous integration	22
6.2.5	System test	22
6.3	Port to new ØMQ library	23
6.3.1	Actual port	24
6.4	Cluster	24
6.4.1	DIM Synchronization	24
6.4.2	Node Typology Definition	26
6.4.3	Message Routing	27
6.5	High Availability	28
6.5.1	Defining Reliability	28
6.5.2	Binary Star in a nutshell	29
6.5.3	Failover	29
6.5.4	Side benefit: Rolling Upgrades	30
6.5.5	Dangerous corner case: Backup node active, dedicated link goes down, and there are new/reconnecting clients	30
6.5.6	Single Level	30
6.5.7	Multi Level	31
6.6	Persistence Synchronization	31
6.6.1	Aspects	31
6.6.2	Variants	31
6.6.3	Chosen Variant	33
6.7	Security	34
6.8	OPC UA Interface: High Availability	34
7	Results	35
7.1	Port	35
7.2	Cluster	35
7.3	High Availability	35
7.4	Persistence Synchronization	36
7.5	Security	36
7.6	OPC UA Interface: High Availability	36
8	Discussion	37
8.1	Value Added	37
8.2	Limitations	37
8.3	Business Benefits	37
8.4	Ideas for Improvement	37
9	Conclusion	39
	Glossary	41
III	Appendix	43
A	Self Reflection	44
B	Task Description	45

C	License	51
D	Project Plan	52
E	ØMQ	53
E.1	Transport Security	53
E.2	Data Serialization	54
E.3	CZMQ	54
F	Infrastructural Problems	55
F.1	Project Management Software	55

List of Figures

4.1	Roadster's software architecture	11
4.2	Roadster's communication layers	11
5.1	Physical legacy example: a single node and a field device each	15
5.2	Physical cluster topology example: supernode, two subnodes, a field device each	15
5.3	Physical cluster topology example: a HA node pair and a field device	17
5.4	Physical cluster topology example: root HA, two subnodes, a field device each	18
6.1	Cluster setup between a supernode and two subnodes	25
6.2	Single level HA setup between a HA pair and a field device (PLC)	30
6.3	Multi level HA setup between a HA pair and a number of client nodes	32

List of Tables

Listings

6.1	Cluster DSL example without HA	26
6.2	Cluster DSL example with HA	26
6.3	Cluster DSL example with HA and roles	27

Part I

Management Summary

Chapter 1

Context

1.1 Initial Situation

Roadster is mindclue GmbH's in-house framework to build modern monitoring and controlling applications in different fields such as traffic systems, energy, and water supply. It is written in Ruby, a modern and expressive scripting language, and is built on a shared-nothing architecture to avoid a whole class of concurrency and scalability issues found in traditional application architectures.

Although considered to be the next generation of its kind, it still lacks important features such as the ability to be run as a cluster on multiple nodes, high availability, and secure network communications.

1.2 Goals

Adding the aforementioned, missing features to form the next version of the framework would mean a distinct advantage for mindclue GmbH and thus increase its competitiveness in its sector.

Planning the exact architectural changes and additions, as well as performing the implementation is the students' goal for this bachelor thesis. Using engineering methodology practiced at HSR, solutions for particular problems will be worked out and the best fitting one will be chosen.

Although not exactly part of the requirements, spreading knowledge about Roadster's architecture and code basis is also in the interest of the client, as Andy Rohr is currently the framework's only developer and thus a single point of failure in an increasingly important piece of software.

1.3 Software development process

The [Rational Unified Process \(RUP\)](#) is used to plan and manage this term project. It's an iterative, structured, yet flexible development process which suits this kind of project. At HSR, it's taught as part of the Software Engineering courses and is thus a primary candidate.

Another candidate was Scrum, which we decided against as it's only feasible with teams of three to nine developers.

1.4 Project management infrastructure

The source code of this document and all of our code contributions are hosted on GitHub. The students will organize and perform their work directly on the site as far as possible. This means creating a Project board for each of the development phases, creating, assigning, and closing issues, as well as using the Wiki feature to plan and document meetings with the professor and the client.

Time tracking, as required by the process for bachelor theses [**hsr:thesis-rules**], are done externally on Everhour.

Chapter 2

Project Phases

TODO describe this phase in retrospection

2.1 Plan

2.2 Inception

TODO describe this phase in retrospection

2.3 Elaboration

TODO include Gantt chart for this phase

TODO describe this phase in retrospection (risk elimination)

2.4 Construction

TODO include Gantt chart for this phase

TODO describe this phase in retrospection (risk elimination)

2.5 Transition

TODO include Gantt chart for this phase

TODO describe this phase in retrospection (risk elimination)

Chapter 3

Results

TODO describe results

Part II

Technical Report

Chapter 4

Scope

The technical goals of this bachelor thesis include extending mindclue GmbH's Roadster framework by adding features such as clustering, high availability and transport security. This chapter outlines the general scope of this project.

4.1 Motivation

Backgrounds

To better understand our motivation, it might help to understand our personal backgrounds first.

Patrik Wenger

Having done his apprenticeship in computer science at Swisscom Schweiz AG, he continued to work as a full-time employee for five years afterwards. In programming he's most fluent in [Ruby](#) and [C](#). During the winter of 2015/2016, he created [CZTop](#) during leisure time because there was no good Ruby binding for [ØMQ/CZMQ](#) available and a side project of his demanded it.

Fascinated with event-driven programming and software design patterns such as the [Actor Model](#)¹, distributed computing and high availability have always been part of his core interests, especially in conjunction with the brilliant [ØMQ](#) library.

Having a passion for information security and state-of-the-art cryptography², especially in this post-Snowden era, this bachelor thesis is like a dream come true.

Manuel Schuler

Always keen on learning new things, he did not hesitate to join this bachelor thesis at the first opportunity.

In essence, we're both thrilled to gain more experience in the following fields and technologies:

- Distributed Computing
- High Availability
- Information Security

¹known from Erlang, brought to Ruby by the Celluloid library, as well as the young programming language Pony which is completely based on actors

²such as [NaCl](#) or [libsodium](#) as used by [ØMQ](#)

- [Actor Model](#)
- [ØMQ](#)
- [Ruby](#)

Opportunities

Coming from different backgrounds and having different levels of experience in each of the above technologies, we can't wait to learn more about them and put them to actual use. The fact that the product of this bachelor thesis is most likely going to be used in the real world only adds to the excitement.

This bachelor thesis involves working with Ruby, the Actor Model, ØMQ, distributed computing with high availability, and state-of-the-art cryptography. Furthermore, in case of successful completion of this thesis, the results will be used in real-world settings like the Ceneri Base Tunnel. It is a huge opportunity for a solution completely based on free and open-source software interacting with other industrial systems over open standards. The students, as well as the client, strongly believe in customized solutions built on reusable, free open-source software.

In addition to that, we look at this bachelor thesis as an opportunity to become more fluent in English, both written and spoken, as well as to improve our skills in crafting scientific documents using \LaTeX .

Depending on how we perform together as a team, further collaboration might result in the future, either between the students themselves, or between the students and the client. Even if our paths will part, this project will serve as a valuable reference for future job hunting.

Last but not least, we feel like Prof. Dr. Mehta is a respected and competent teacher whose opinions we highly value. Due to his polite parlance, discussing project matters, both of the management and the technical kind, has always been an enrichment.

4.1.1 Open-Source engagement

Getting the chance to use [CZTop](#) and watch it perform definitely adds to the motivation as well. Its software design has yet to be proven in more serious settings.

Another personal goal is to create a reusable open-source library as a byproduct. The intention is that the library makes certain ØMQ-based communication protocols readily available for other developers facing the same problems.

4.2 Initial Situation

4.2.1 mindclue GmbH

The company mindclue GmbH, located in Ziegelbrücke GL, provides its partner REMTEC AG with complete [Supervisory Control and Data Acquisition \(SCADA\)](#) applications. These are then used to control and monitor operation and safety equipment found in national freeways, water supply systems, as well as in energy facilities and many other specialized fields. To build these customized applications, their in-house creation Roadster, a next-generation SCADA framework, is used.

4.2.2 Roadster

Roadster is a SCADA framework written in Ruby. It was, and still is, developed to produce next-generation SCADA applications to replace legacy solutions based on its predecessor found in numerous tunnel facilities in Switzerland.

A Roadster installation combines the following responsibilities:

- interaction with subordinate field devices (monitoring & controlling)
- persisting data (e.g. certain sensor data, and events)
- sophisticated alarm (*case*) management
- providing an interface ([Open Platform Communications \(OPC\) Unified Architecture \(UA\)](#)) to superordinate systems
- providing a modern web UI for interaction with operational and executive personnel

Among others, subordinate field devices include [Programmable Logic Controllers \(PLCs\)](#) and emergency information systems. These are interacted with over numerous proprietary and/or standardized protocols. Superordinate systems interact with Roadster over protocols including [Service Oriented Application Protocol \(SOAP\)](#) and [OPC UA](#). Their purpose is to collect and aggregate information from larger regions. At the top of the hierarchy are the ASTRA and MINSTRA which combine the information of all subsystems and provide a nationwide overview.

4.2.2.1 Typical hardware

Roadster typically runs on entry-level rack server hardware powered by an Intel® Xeon® processor, or industrial box PCs for smaller systems commonly used for [Internet of Things \(IoT\)](#) which are powered by more energy efficient processors such as Intel® Core® and Intel® Atom™. The machines are usually equipped with 4 – 6 GiB of main memory and Gigabit Ethernet. For reliable systems without any moving parts, one industrial grade [Solid State Disk \(SSD\)](#) or two (in a software [redundant array of independent disks \(RAID\)](#) level 1) setup are used.

4.2.3 ØMQ

To understand Roadster's architecture and the rest of this document, it's helpful to understand the basics of ØMQ first. This is a brief introduction to ØMQ for the unfamiliar reader. What follows is a quote from the [Zguide](#) which does a fairly good job at describing ØMQ in a 100 words:

“ZeroMQ (also known as ØMQ, 0MQ, or zmq) looks like an embeddable networking library but acts like a concurrency framework. It gives you sockets that carry atomic messages across various transports like in-process, inter-process, TCP, and multicast. You can connect sockets N-to-N with patterns like fan-out, pub-sub, task distribution, and request-reply. It's fast enough to be the fabric for clustered products. Its asynchronous I/O model gives you scalable multicore applications, built as asynchronous message-processing tasks. It has a score of language APIs and runs on most operating systems. ZeroMQ is from iMatix and is LGPLv3 open source.”

Roadster uses ØMQ to carry messages between its processes.

For a more detailed introduction, see [Appendix E](#).

4.2.4 Software architecture

As mentioned earlier, Roadster is event-driven³ and built on the Actor model, meaning it exhibits a shared-nothing architecture. Each Roadster node runs a number of Ruby processes which communicate via ØMQ sockets. The key here is communication:

“Don’t communicate by sharing state; share state by communicating.”

Running multiple, loosely coupled processes (actors) allows leveraging the full potential of modern multi-core processors, while avoiding a whole class of traditional concurrency problems.

Every Roadster node runs a group of actors:

CORE: It is responsible to start the other actors. It also plays a key role in keeping state in all actors synchronized, being the source of truth.

COMM: A bunch of COMM actors communicate with the outside world of a node. It typically either acts as a client of various kinds of subordinate field devices, or as a server to superordinate systems. To communicate with subordinate systems, a COMM actor uses an adapter specifically written for the communication protocol in place.

STORAGE: This actor is used when information needs to be persisted, such as time series or event journals. It’s the interface to a key-value store.

LOGGER: This actor collects logging data and sends it to whatever target is configured, be it STDOUT, a file, or a syslog server.

Figure 4.1 illustrates Roadster’s architecture.

4.2.4.1 Communication Layers

The communication architecture in Roadster consists of three layers, as illustrated in Figure 4.2. The following list briefly explains the layers from top (most abstracted) to bottom:

Engine layer:

Here is the business logic of Roadster, e.g. the [Domain Information Model \(DIM\)](#), user authentication, adapters for different devices, the web [user interface \(UI\)](#), etc.

Messaging layer:

The [Roadster Messaging Protocols \(RMP\)](#) reside here and implement essential protocols used for logging, state synchronization, commands, application controlling, and storage. They’re explained below in [subsubsection 4.2.4.2](#).

Reactor layer:

This layer forms the base, which is where the ØMQ sockets and WebSockets used. In case of COMM actors, there can also be raw TCP sockets, e.g. to interact with certain [PLCs](#).

4.2.4.2 RMP

The [RMP](#) are a collection of protocols implemented and used by Roadster internally. They reside in the messaging communication layer, and include:

Clone State Protocol (CSP):

Used to synchronize state between the actors.

Application Control Protocol (ACP):

Used to control the application state, e.g. things like shutdown.

³https://en.wikipedia.org/wiki/Event-driven_programming

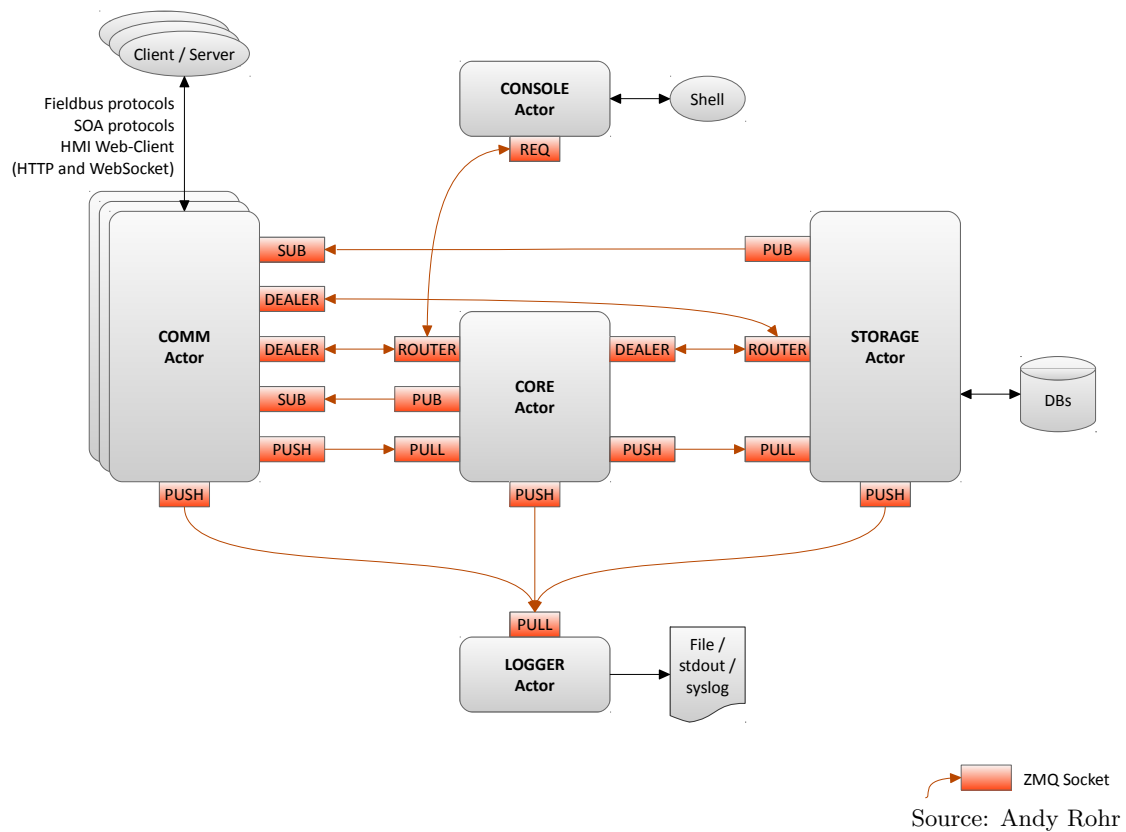
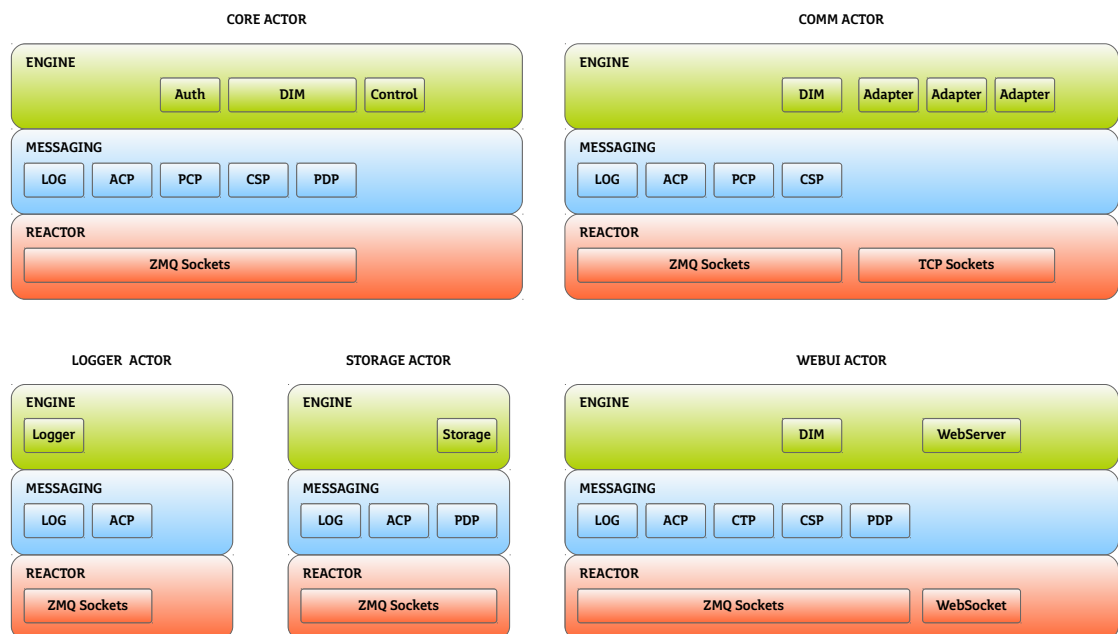


Figure 4.1: Roadster's software architecture



Source: Andy Rohr

Figure 4.2: Roadster's communication layers

Persistent Data Protocol (PDP):

Used when data needs to be persisted.

Supress Management Protocol (SMP):

Used to suppress the generation of certain [cases](#), e.g. when a sensor is defect and repeatedly causes cases.

Peer Control Protocol (PCP):

Used for asynchronous command execution via COMM peers with feedback.

LOG protocol:

Used for system logging.

Every actor in Roadster uses a subset of these protocols to perform its job.

Passing messages from actor to actor, which are nothing but serialized Ruby objects, happens in one of two modes:

Fire & Forget: No guarantee of correct processing, e.g. DIM updates from COMM to CORE. This doesn't mean there are no other mechanisms in place to ensure reliability.

Dialog: An immediate answer is expected, e.g. when creating a user session. Sending a message like this looks like it's a synchronous call, even though it's handled asynchronously under the hood⁴. Any protocol can make use of this primitive.

4.2.4.3 DIM

The [DIM](#) is a data structure that lives inside every actor of a Roadster node. Every actor builds it when starting up by reading the configuration files. Updates to certain parts⁵ within it are replicated across all actors. An updated item is marked dirty⁶ so it is subsequently synchronized via the [CSP](#).

4.2.4.4 Existing CSP in a nutshell

This is a brief introduction/refresher for the Clone State Pattern implemented by Roadster, which is used for the DIM synchronization. Although Roadster actually sends serialized instances of CSP message classes to fulfill this protocol, for better readability the [Zguide](#)'s canonical nomenclature of [Clone Pattern](#) messages will be used.

The existing [CSP](#) is closely related to the [Clone Pattern](#) from the [Zguide](#). Its goal is to keep a state (a list of key-value pairs) in sync across a set of participants. To greatly reduce the complexity, it's not decentralized: There's a server part which serves as the single source of truth.

The server uses a ROUTER, a PULL, and a PUB socket; each client a DEALER, a PUSH, and a SUB socket. The protocol consists of three distinct message flows:

Snapshots: Requesting and receiving the complete, current snapshot of the state (all key-value pairs). This happens via a ROUTER/DEALER pair of sockets. The request message consists solely of the humorously named ICANHAZ command. The response is the complete set of KVSET messages so a late-joining (or previously disconnected) client can rebuild the current snapshot.

⁴This is done by wrapping the affected code in a Ruby [Fiber](#), which is similar to a thread but allows for cooperative scheduling as opposed to preemptive.

⁵Namely instances of the meta-model classes [Case](#), [DataItem](#), and [Session](#)

⁶It is marked dirty by setting its `@lifecycle_state = "updated"`

Upstream updates: Updates always originate from clients and are sent to the server via a PUSH/PULL pair of sockets. These are KVSET messages.

Downstream updates: After being applied to the server's copy of the state, updates get a sequence number and are published back to all clients. This happens via the PUB socket and uses KVPUB messages.

By making all updates go through the server, a total order is enforced, which is crucial to keep the state consistent across all clients.

To avoid risking a gap between requesting the current snapshot and subscribing to updates, a client actually subscribes to the updates first, then gets the snapshot, and then starts reading the updates from the socket (which has been queueing updates in the meantime, if any). Updates that are older or the same age as the received snapshot are skipped, and only successive updates are applied (tested by comparing the sequence numbers).

Because message loss via the third message flow (PUB-SUB) is unlikely but theoretically possible, the client checks for gaps in the sequence number of each KVPUB message. If a gap is detected, the current state is discarded and a complete resynchronization happens. This is brutal, but is very simple and thus robust; there's no complexity that would leave room for nasty corner cases.

Keys can be treated hierarchically (e.g. `topic.subtopic.key`) and thus, a client can optionally subscribe to only a particular subtree. This is useful when the number of client grows and not all of the state needs to be on every client. In that case, the topic of interest is sent by the client along with the ICANHAZ message.

4.3 Goals

To summarize the mandatory goals from the Task Description in [Appendix B](#):

1. Getting familiar with Roadster
2. Extending the communication protocols to support clustering
3. Extending the communication protocols to add high availability

The optional goals are:

1. Encryption of the communication
2. Providing of the highly available [OPC UA](#) server interface

Secure inter-node communication within a Roadster cluster is important to mitigate common security concerns with SCADA systems which are becoming more and more open due to standardization. To quote wikipedia:

“In particular, security researchers are concerned about:

- the lack of concern about security and authentication in the design, deployment and operation of some existing SCADA networks
- the belief that SCADA systems have the benefit of security through obscurity through the use of specialized protocols and proprietary interfaces
- the belief that SCADA networks are secure because they are physically secured
- the belief that SCADA networks are secure because they are disconnected from the Internet.”

Chapter 5

Requirements

The requirements gathered during the first meeting with the client are explained in this chapter. These are more concrete than the ones listed in the Task Description in [Appendix B](#).

First of all, these are the priorities from the client's point of view in descending order:

1. Clustering
2. Single-level [high availability \(HA\)](#)
3. Multi-level [HA](#)
4. Persistence synchronization
5. Security (optional)
6. OPC UA [HA](#) (optional)

The following sections explain the requirements in greater detail.

5.1 Functional

This section elucidates the functional requirements, as opposed to the non-functional requirements.

5.1.1 Cluster

Roadster must be able to run on multiple nodes in a hierarchical topology, forming a distributed computing cluster. Typical node topologies include:

Single level, single node

This is the legacy setup and is what Roadster is already able to do. Consisting of only one node, it's not actually a cluster. This is illustrated in [Figure 5.1](#).

Multi level

This is the most basic cluster setup. There is a root node, and two subnodes. Each subnode is directly connected to a field device such as a PLC or an emergency phone. This is illustrated in [Figure 5.2](#).

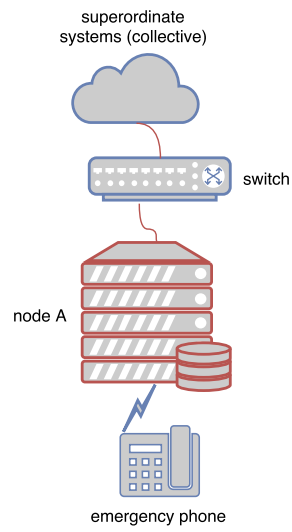


Figure 5.1: Physical legacy example: a single node and a field device each

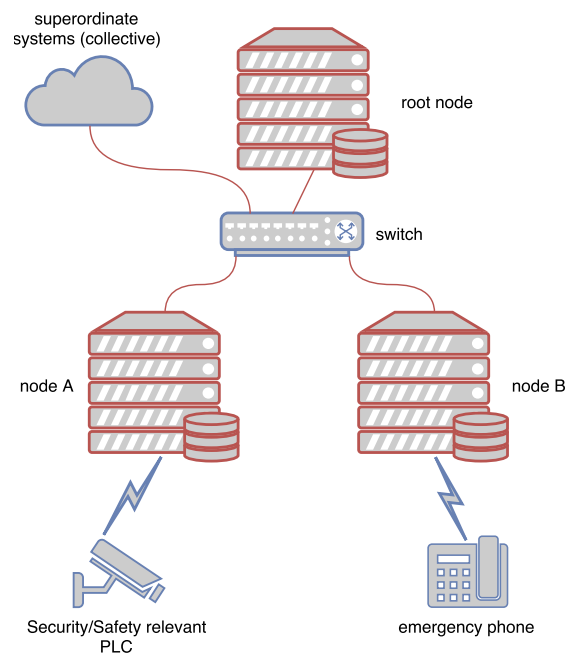


Figure 5.2: Physical cluster topology example: supernode, two subnodes, a field device each

5.1.1.1 DIM synchronization

Extending the [CSP](#) to keep the [DIM](#) in sync across all nodes is a central part of the clustering functionality. This means replicating the items within the DIM that are marked "dirty" (updated, but not synchronized yet) onto all other actors on all other nodes.

According to the Data Model class diagram, these should only be instances of one of three classes (marked yellow in the diagram):

- [DataItem](#)
- [Session](#)
- [Case](#)

5.1.1.2 Message routing

In addition to that, there needs to be a message routing mechanism so a user of one node's web UI can send a command to another node where it will be executed. An example for this is a forced value in the DIM to ignore the actually measured value reported by a device in case the device is known to be wrong. The common case where the command is issued at a higher level in the node topology is priority. E.g. in a setup with a root node and two subnodes A and B, issuing a command on A for B has a low priority.

5.1.1.3 Autonomy

It's important that every node subtree can keep up the operation autonomously even if the link to its supernode or the supernode itself fails. This means that updates to the [DIM](#) are allowed even when the supernode is unavailable. After the recovery from the outage, the [DIM](#) synchronization shall be reinitiated so all pending updates are shared to all other nodes (through the supernode).

5.1.1.4 Access restriction

The above requirements imply that changes to the [DIM](#) can only be done by the respective node. In other words, a node can only change its own values. It cannot change values of supernodes, nor is it allowed to change values of subnodes directly. This is to ensure that each node is its own source of truth to all other nodes in the setup. Only the responsible node can enforce a single sequence of updates to its part of the DIM, which is necessary to guarantee consistency [6, Chapter 5, Reliable Pub-Sub (Clone Pattern), Republishing Updates from Clients].

5.1.2 High availability

Roadster must be able to run in certain high availability setups. Achieving this is done by adding redundant Roadster nodes. There must be two distinct [HA](#) modes available: Single level and multi level. These determine at which level (at the bottom or at root of the topology) redundancy is added.

The following additional topology setups must be supported:

Single level [HA](#)

This is when there are exactly two nodes, both of them connected to the same PLC. There are two nodes for redundancy. This is illustrated in [Figure 5.3](#).

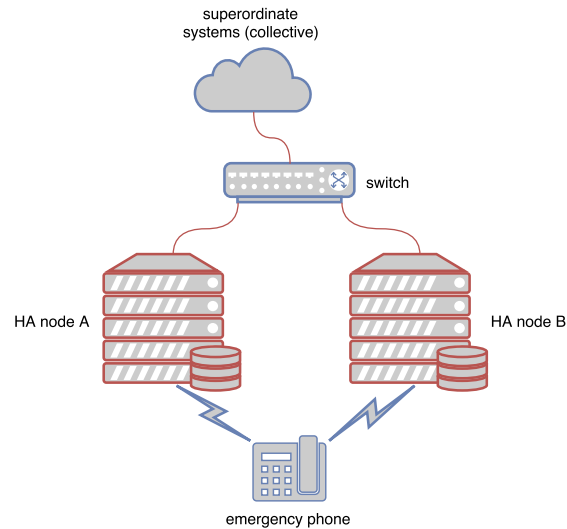


Figure 5.3: Physical cluster topology example: a HA node pair and a field device

Multi level, HA at root only

There can be multiple levels in the hierarchy, such as two or three (anything else is considered exotic), and there's a HA cluster at the root. An example of this setup is illustrated in Figure 5.4.

The following exotic cases can be ignored:

Multi level, HA at bottom

A single root node at the top and a subordinate HA pair each connected to the same field device.

Multi level, HA in the middle

A single root node at the top, a subordinate HA pair, which in turn has a subordinate node connected to some field device.

The following sections contain the concrete requirements about the two different HA levels.

5.1.2.1 Single level

This is where each of the two nodes forming a HA unit is directly connected to a number of subsystems such as PLCs, forming two redundant network paths to each subsystem. Both nodes are able to interact with the subsystems to perform operation tasks (e.g. reading sensor data, writing down configurations), but only one of them (the active one) must do so.

The two nodes must automatically find consensus on which one is currently active. The passive one must automatically take over in case the active one is confirmed to be dead. Certain memory ranges can be used by the nodes to help find consensus without interfering with normal operation of the subsystem. The kinds of failures that need to be handled include:

- Hardware/software failure on the primary node
- Failure of one of the redundant networking paths connecting the subsystem to the two nodes

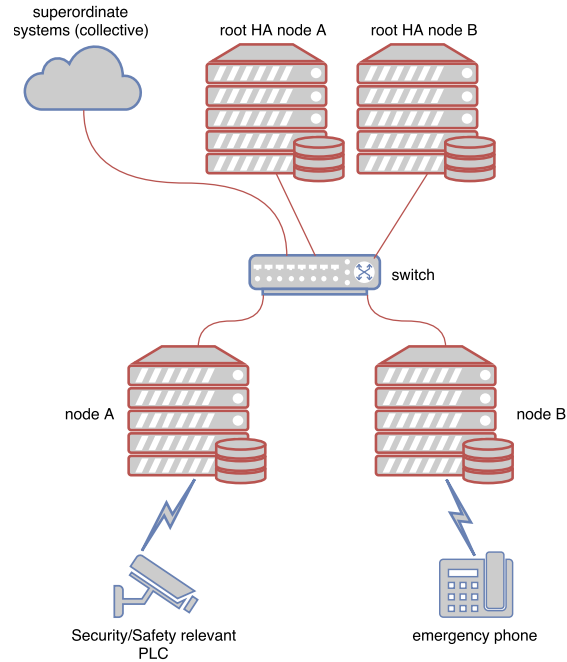


Figure 5.4: Physical cluster topology example: root HA, two subnodes, a field device each

5.1.2.2 Multi level HA

This is where a node pair is the parent of one or more subnodes.

The types of failures that need to be handled include:

- Software failure on the primary node, like an application or OS crash
- Hardware failure on the primary node, like a defect power supply
- Failure of the network link connecting a node to the cluster

All three failure types listed above can collectively be called *crash*, as their effects are the same from the point of view of the cluster.

5.1.3 Persistence synchronization

This is about the synchronization of persisted data, which is currently stored in TokyoCabinet databases on a Roadster node. With the clustering, this is still true: Every node will have its own key-value store. Updates for persisted data must only flow from south to north (towards the root node), so the root node can collect and maintain a replication of the persisted data of all subnodes, recursively.

It's important that every node and its subnodes form an autonomous subtree. So in case the link to its supernode fails, it has to continue working. As soon as the link is repaired, synchronization of the delta (the newly added data) can be initiated. Sending just the delta should be possible since keys in the database contain timestamps.

This is not the same as [Clustered Hashmap Protocol \(CHP\)](#) ([DIM](#) synchronization), as the DIM is shared across all nodes and is a relatively small data structure. The TokyoCabinet databases can possibly contain large amounts of data (in the hundreds of megabytes) and are shared only towards the root node (thus "bubbling up").

100% consistency is not an absolute requirement for persistence synchronization. However, it is mandatory that updates make it to the root node within 30 seconds.

5.1.4 OPC UA HA

This is optional.

A given HA pair needs to provide an OPC UA Server Redundancy interface, as described in [4, 6.4.2.4 Non-transparent Redundancy, p. 96].

5.2 Use Cases

The use cases are briefly described here. These should simply describe common scenarios based on the requirements above. They can later be turned into concrete testing scenarios.

5.2.1 UC01: Cluster

A root node R has two subnodes A and B. A user connected to the root node's web UI wants to suppress a case repeatedly generated on subnode A. The DIM shall be kept in sync across all nodes.

5.2.2 UC02: Hardware failure at top

A subnode A is connected to a root-level HA pair (nodes R1 and R2). The active HA peer (R1) crashes. R2 is supposed to take over as soon as subnode A fails over to R2.

5.2.3 UC03: Hardware failure at bottom

A HA pair is connected to a field device. The active nodes crashes. The passive one has to notice and take over to ensure continued monitoring of and access to the field device.

5.2.4 UC04: Persistence synchronization

A root node R has two subnodes A and B. Persisted data on A has to bubble up to root, even after temporary failure of the link between them. Root eventually contains the union of all subnodes' persisted data.

5.2.5 UC05: OPC-UA HA

A HA pair provides an OPC UA interface. On failover, the superordinate system shall continue to interact with the leftover HA peer.

5.3 Non-Functional Requirements

The following subsections illustrate the non-functional requirements.

5.3.1 Simplicity

The two reoccurring patterns that surfaced during the requirements gathering meeting were:

1. **KISS** principle. Simplicity is favored, as experience shows that simpler systems are more stable, so complexity should be avoided if not absolutely necessary.
2. No premature optimization since it's the root of all evil.¹

5.3.2 Testing

Regarding testing, the following requirements exist:

- the student's contributions are verified with at least unit tests
- use cases shall be integration tested in a close-to-reality setup

5.3.3 Security

This is optional. Also, this requirement has the lowest priority not because it's insignificant, but because it's easy to enable transport level security on ØMQ sockets.

The inter-node communication of a Roadster cluster must be secured using encryption. ØMQ offers server authentication at a minimum, if encryption is enabled. Client authentication is not needed at this stage.

5.3.4 Coding Guidelines

The coding guidelines desired by the client are basically the ones written down in the popular Ruby style guide [1], with the following differences or special remarks:

- method calls: only use parenthesis when needed, even with arguments (as opposed to ²)
- 2 blank lines before method definition (slightly extending ³)
- YARD API doc, 1 blank comment line before param documentation, one blank comment line before code (ignoring ⁴)
- Ruby 1.9 symbol keys are wanted (e.g. `foo: "bar", baz: 42` instead of `:foo => "bar"`, `:baz => 42`, just like ⁵)
- align multiple assignments so there's a column of equal signs

¹Quote by Donald Knuth: "Premature optimization is the root of all evil."

²<https://github.com/bbatsov/ruby-style-guide#method-invocation-parens>

³<https://github.com/bbatsov/ruby-style-guide#empty-lines-between-methods>

⁴<https://github.com/bbatsov/ruby-style-guide#rdoc-conventions>

⁵<https://github.com/bbatsov/ruby-style-guide#hash-literals>

Chapter 6

Approach

TODO what have we done to arrive at the goal (should be reproducible)

TODO this is probably what we know as "Concept"

6.1 Getting familiar with Roadster

The client gave a short introduction into Roadster's code base during the first week of this thesis. Although quite overwhelming, the first impression was that the code is clean, makes use of useful abstractions and has loosely coupled classes. API documentation is scarce though.

TODO: describe approach of getting more familiar with Roadster, e.g. during prototyping

6.2 Testing

This section describes the test methods we used to check particular methods, the integration of multiple components, as well as the behavior of the whole application. All test results can be found in [chapter 7](#).

6.2.1 Setup

Due to the fact that Roadster's Github repository is private, online [continuous integration \(CI\)](#) services such as *Travis CI*¹ can't be used without payment². Fortunately, Gitlab CI is free and can be installed on one's own infrastructure, such as the [virtual machine \(VM\)](#) provided by HSR, where it was installed and configured. Unit and integration tests are run every time new commits are checked in. This is useful to get informed proactively when something breaks.

¹[urlhttps://travis-ci.com/](https://travis-ci.com/)

²Payment is required after the first 100 builds.

6.2.2 Unit tests

To ensure the correctness of the implementations, unit tests are written using RSpec. 100% coverage of the students' contributions can be achieved by adhering to [test-driven development \(TDD\)](#). Naturally, this also simplifies refactoring the code without risking things breaking silently. Unit tests reside under the `spec` directory of Roadster's code base.

6.2.3 Integration tests

Integration tests verify the interaction between the individual components. To test core features like cluster, high availability, and persistence synchronization, integration tests have been written. Multiple nodes can be simulated easily by starting them as different process groups. This is possible since ØMQ completely abstracts the transport away.

In order to test the failover or synchronization functionality, individual processes can simply be killed and restarted later if the scenario defines this.

More details about the integration test scenarios can be found in [chapter 7](#).

6.2.4 Continuous integration

[CI](#) helps us prevent integration problems also known as *integration hell*. Each push to the repository will trigger a CI check, which will run a build script. This will install Roadster's dependencies, an example app built with Roadster, and finally Roadster's test suites.

6.2.5 System test

Systemtests dienen dazu die Applikation unter realen Bedingungen zu testen. Um eine fast realitätsnah Umgebung zu erschaffen, kam mininet, jnettop und fake SPS Steuerungen zum Einsatz.

Mininet: Mininet erlaubt es adhoc VirtuelMachine's zu starten, welche einen gemeinsamen Kernel teilen. Dies ermöglicht es auf einem Rechner viele Nodes/Rechner zu starten. Mittels mininet können auch Verbindungen zwischen Nodes getrennt werden um Netzwerkprobleme zu simulieren.

jnettop: jnettop simuliert Netzkapazitätsprobleme auf den einzelnen Verbindungen.

fake sps: Fake SPS Steuerungen sind einfache Scripts, welche auf Anfragen antworten.

Alle Ereignisse wurden in Logdateien gespeichert, welche für die Auswertung des Ergebnis gebraucht wurden. Mit einem Ruby Programm wurden diese Logdateien auf Korrektheit überprüft. Alle Systemtests wurden manuell nach jeder construction phase ausgeführt.

System tests are designed to test the application under real conditions. To create an almost realistic environment we used mininet, jnettop and fake [PLC](#) controls for deployment.

Mininet: Mininet allows adhoc start VirtuelMachine's, which share a common kernel. This makes it possible on a computer to start many nodes / hosts. Mininet can also simulated connection problems between nodes.

jnettop: jnettop simulated network capacity problems on the individual network connections between the nodes.

fake sps: Fake [PLC](#) controls are simple scripts that respond to requests.

All events were stored in log files, which were used for the evaluation of the result. A Ruby program checked these log files on correctness. All system tests were performed manually after each construction iteration.

6.2.5.1 Test scenarios

Die Testszenarien enthalten primär nur mögliche Bediengungen wie sie in einer realen Situation anzutreffen sind. Aus wissenschaftlichen Interesse wurden weitere Szenarien getestet, welche den Extremfall abbilden sollten.

The test scenarios only includes possible combinations as they are encountered in a real situation. From scientific interest we added more test scenarios, which reflected the extreme cases.

ML-HA: Multi Layer mit zwei ebenen. Die erste Ebene enthält ein Node, welches die Steuerungen überwacht. Die zweite Eben hat einen Primary und Backup.
TODO ...

SL-HA: Single Layer. Ein Primay und Backup Node welche beide direkt mit der Steuerungen verbunden sind.
TODO ...

Persistence wird bei allen Tests mit geprüft, da sie zu Non functional requirements gehört und bei allen Szenarien aktiv ist.

TODO Extremfall ... 3 Ebenen mit 1 ebe

Testszenarien enthalten die Konfiguration von Mininet und den einzelnen Nodes. Der Ablauf der Szenarios wurde im Detail beschrieben um die Ergebnisse reproduzierbar zu machen.

TODO work out methodology (maybe with mininet, shell scripts, and analyze logs retrospectively)

TODO integration tests at the end of every iteration

TODO system tests at end of construction

TODO use travis-ci.com OR GitLab CI (self-hosted on HSR VM)

6.3 Port to new ØMQ library

Porting Roadster to a new ØMQ library early on makes sense for the following reasons:

- to exclude possible failures from faults in the unmaintained ffi-rmq³ library
- encryption is needed later anyway, which is not supported by the currently used library
- all other tasks involve ØMQ communication anyway

There is currently only a single Ruby library that is maintained, supports encryption, and freely available, which is [CZTop](#). Technically it's a binding for the [CZMQ](#) library, which is the modern and recommended way of using ØMQ. More info about CZMQ can be found in [Appendix E](#).

As stated in the Task Description already, Roadster's event loop makes use of the ØMQ options ZMQ_FD and ZMQ_EVENTS. Getters for these had to be added to in CZTop, which was a matter of minutes.

³<https://github.com/chuckremes/ffi-rmq>

6.3.1 Actual port

Due to Roadster's beautiful software architecture, code that actually made use of the ffi-rmq library directly was located in a single file. The following things needed to be done:

- Tell Ruby to load CZTop instead of ffi-rmq.
- Remove code to send and receive multi-part messages. This has been simplified in CZMQ and thus is a single method call using CZTop.
- Remove error checking code. CZTop always checks error codes, and raises an appropriate exception if needed.
- Simplify code that reads option values such as ZMQ_FD and ZMQ_EVENTS.
- Rewrite library calls to use CZTop instead of ffi-rmq.

This was about an hour's work.

6.4 Cluster

A Roadster cluster is illustrated in [Figure 6.1](#). Adding cluster functionality to Roadster involves the following aspects:

- node topology DSL
This DSL also has to provide means to define the roles/functionality of each node, e.g. the set of COMM actors running on a particular node
- DIM synchronization
- message routing
- What needs to be done if a WebUI user wants to e.g. change some value on a PLC, possibly on a remote node? Is it completely handled via DIM or do we need message routing?

6.4.0.1 Fallacies of Distributed Computing

At this place, it is worth noting the common fallacies encountered in distributed computing, as explained on [\[5\]](#).

6.4.1 DIM Synchronization

The following is a list of things that are missing before the requirements can be fulfilled:

- it has to work across several nodes
- it has to be able to handle HA supernodes, which affects DIM synchronization and message routing

There are multiple choices when it comes to what exactly of the DIM should be synchronized:

Variant 1. Sync self-subtree only

Always sync on subtree only, which means a node only knows the DIM part of itself. The big disadvantage is that it won't have a copy of the rest of the DIM, which can be useful to inspect variables on neighboring nodes, especially when they're unreachable.

Variant 2. Sync complete tree

6.4.1.1 CAP theorem

The CAP theorem [2] states that it is impossible for a distributed computer system to simultaneously provide consistency, availability, and network partition tolerance. In the face of a network partition, one has to choose between availability and consistency. Because subtrees of a Roadster cluster must be autonomous, availability is chosen.

Eventual consistency is guaranteed by restricting write access to the owning node, and recovering from a network partition when communication is restored is done by simply reinitiating the DIM synchronization process.

6.4.2 Node Typology Definition

A cluster's node topology has to be defined somewhere. This can be done using a [Domain Specific Language \(DSL\)](#) and then put into a static file (e.g. `topology_conf.rb`) shared on all nodes of a Roadster cluster. Each actor could then read the file at startup, just like it's done for other configuration pieces of a Roadster node. [Listing 6.1](#) shows how such a configuration snippet might look.

To let the actors of a node know which node they belong to, an additional line has to be added to the specific configuration file (`conf.rb`), e.g. `conf.system_id = "nodes.root"`. Using that information, the topology created using the DSL can be walked like a tree to find the correct node and important information like its neighbor nodes.

A HA node pair could be one DIM object which has one name but two IP addresses (primary and backup, in that order). Direct subnodes can use that information to connect to the correct superordinate node during normal operation and also when the primary node is unavailable. The respective DSL snippet is shown in [Listing 6.2](#).

Not every node in a cluster is the same: Some are only connected with other nodes, some also talk to field devices. To define different node roles, a syntax as shown in [Listing 6.3](#) is possible.

```
# * basic method to add a node: #add_node(ID, south_facing_bind_endpoint)
# * it takes a block for defining subnodes

conf.nodes do |map|
  map.add_node("root", "tcp://10.0.0.1:5000") do |map|
    map.add_node("subnode_a", "tcp://10.0.0.10:5000")
    map.add_node("subnode_b", "tcp://10.0.0.11:5000")
  end
end

# subnode_a can infer its endpoints from its position in the tree:
conf.system_id = "nodes.root.subnode_a"
# => this node is "subnode_a"
# => its IP address is 10.0.0.10
# => north facing COMM actor's bind port is 5001
# => south facing COMM actor's bind port is 5000
# => north facing COMM actor will connect to "root" node on "tcp://10.0.0.1:5000"
```

Listing 6.1: Cluster DSL example without HA

```
conf.nodes do |map|
  map.add_ha_pair("root", "tcp://10.0.0.1:5000", "tcp://10.0.0.2:5000") do |map|
    map.add_node("subnode_a", "tcp://10.0.0.10:5000")
    map.add_node("subnode_b", "tcp://10.0.0.11:5000")
  end
end
```

```

end

# subnodeA can infer its endpoints from its position in the tree:
conf.system_id = "nodes.root.subnode_a"
#=> this node is "subnode_a"
#=> its IP address is 10.0.0.10
#=> north facing COMM actor's bind port is 5001
#=> south facing COMM actor's bind port is 5000
#=> north facing COMM actor will connect to "root" HA pair on "tcp←
    ↳ ://10.0.0.1:5000" OR "tcp://10.0.0.2:5000" (Lazy Pirate algorithm)

# for primary root:
conf.system_id = "nodes.root[primary]"

```

Listing 6.2: Cluster DSL example with HA

```

# Idea for node topology definition and assigning roles (features/adapters) to
# diffent kinds of nodes.

module Roadster
  module Domain::Model

    build do
      nodes do
        node "root" do # or maybe ha_node or bstar_node
          endpoint "tcp://10.0.0.1:5000", "tcp://10.0.0.2:5000"
          label 'BA Roadster App'
          desc 'Sample application for experimenting and developing the new ←
              ↳ features within the scope of the Bacherlor Thesis of Patrik ←
              ↳ Wenger and Manuel Schuler at HSR.'

          load_conf ::Conf::AccessControl
          load_conf ::Conf::Objects
          load_conf ::Conf::Navigation

          node "subnode_a" do
            endpoint "tcp://10.0.0.1:5000"
            load_conf ::Conf::Adapters
            # load_conf ...
          end
        end
      end
    end

  end # Domain::Model
end # Roadster

```

Listing 6.3: Cluster DSL example with HA and roles

6.4.3 Message Routing

Messages need to be sent from an actor on one node to an actor on another node. The best place to put this logic is the CORE actor which already does this for messages exchanged within a node. It needs to be extended to know about nodes and their actors, not only actors on the current node. Then messages can be passed around hop-by-hop.

In case a message is sent in *Dialog* mode, this implicates Russian doll routing: At every hop, a new dialog is started which expects an immediate response, which will subsequently be passed back and complete the open dialogs.

6.4.3.1 Example

When a user of the root node's web UI wants to change a value on a field device connected to the root node's subordinate node, a command is sent from the browser to the web UI's COMM actor. From there it's sent via the CORE actor out on the south-facing CLUSTER actor to the subordinate node. There it's routed via the CORE actor to the correct COMM actor, where the command can actually be executed on the field device.

6.5 High Availability

If Roadster is going to be run in a cluster setup, measures need to be taken to mitigate the risk of failure, since many nodes are more likely to fail than a single node (unless they add redundancy). Availability shall be ensured by adding redundancy on certain levels of the node hierarchy (e.g. at the bottom of the topology, right above the PLC, or at the root level), in the form of a fully functional backup node in addition to the primary one.

Run together in a hot-standby cluster, the passive node's responsibility is to take over in case the active one goes down.

6.5.1 Defining Reliability

When speaking about reliability, it's worth listing the failures we want to be able to handle. According to the requirements, these are exactly:

Hardware or software failure on the primary node: This could be one of the actors crashing, the whole OS crashing, or a fatal disk failure, irrecoverable memory error, or even just someone accidentally pulling the power plug.

Network failure This only includes the failure of the link connecting a HA node to the rest of the cluster. Interestingly, this limitation applies to both single level and multi level HA.

Failures that won't be covered include:

Failure of the link between a subnode one of its supernodes: This can't be handled since the two HA peers would have to continually share the number of subnodes connected to them, and based on that, make a decision on which one should be active or passive. Since the link between them could fail as well, this decision can't be done reliably, which could lead to the dreaded split brain syndrome.

Failure of the link between a HA peer node and the field device The [Binary Star Pattern](#) algorithm won't initiate a failover since the active is still alive and is able to tell the passive node so. The missing life signs via the field device could cause an alarm, but no failover, since they're only half of the conditions that have to be met for a failover.

The [Zguide](#) describes a very simple mechanism to achieve this kind of high availability with exactly two redundant nodes: The [Binary Star Pattern](#). It provides a set of clients a highly available service by running two server nodes in a hot-standby setup. It is simple and thus very robust, avoids the split-brain syndrome, and is fairly easy to implement, even as reusable code. The implementation could be contained within a new kind of COMM actor called BSTAR. This makes sense since it talks to the outside world.

6.5.2 Binary Star in a nutshell

Two HA peer nodes are started either as primary or as backup. After an initial handshake, the primary one becomes active, the backup node becomes passive. The two continually exchange heartbeats. Clients always connect to the primary's endpoint first.

The passive node takes over when the following two conditions are met:

1. no life signs from the active node
2. connection requests from clients

The second condition is to prevent the split-brain syndrome and thus can be thought of as an external vote for the node to actually initiate the failover. This works because clients will always try to connect to the primary node's endpoint first, then move on to the backup node's endpoint. This algorithm is explained in [6, Chapter 4 - Reliable Request-Reply Patterns, Client-Side Reliability (Lazy Pirate Pattern)].

6.5.3 Failover

In case the currently active node goes down, the two conditions will be met. This means that the passive node starts accepting snapshot requests (ICANHAZ messages) and updates the DIM, so every other node will know about the new, active node. This is needed for the message routing to work.

It's important to mention that a dedicated, direct link from one HA node to its peer actually worsens high availability. In case the non-dedicated link from the primary HA node goes down, meaning the HA node is effectively offline and unavailable for subnodes, the failover won't happen since heartbeats are still exchanged with the HA peer node over the dedicated link.

6.5.3.1 Alarm Generation

When a failover happens, it makes sense to create a [Case](#) (alarm) in the DIM, so the outage is visible to operational personnel in one of the web UIs. The same applies to the case where the passive node goes down, although it doesn't have an immediate effect on availability. This is so the operational personnel can act upon the alarm and e.g. initiate field forces to inspect the failed node and repair it.

Once repaired, it's restarted with the exact same configuration — either primary or backup. Since there's already an active node (either the primary one, or the backup one), the newly repaired node will become the new passive node.

6.5.3.2 Failover from Backup to Primary

Once failed over, the newly active backup node stays active. It does so until it fails itself, at which point the now repaired, up and running primary node will take over. This works because the [Binary Star Pattern](#) operates symmetrically after successful initialization. But it never automatically switches back to make the primary node the new active one without a failure. This is key. If a node goes down, the failover happens automatically, but anything else will require human interaction.

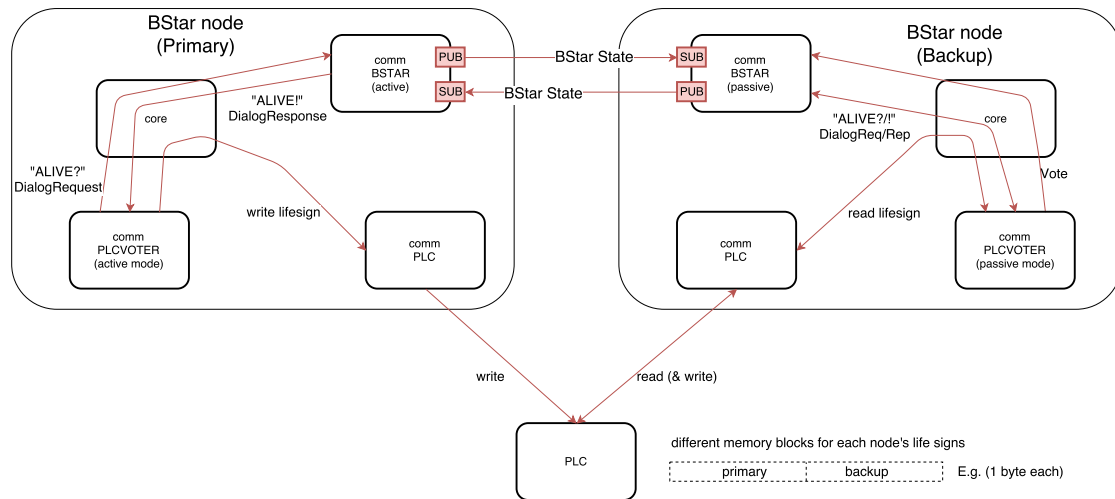


Figure 6.2: Single level HA setup between a HA pair and a field device (PLC)

6.5.4 Side benefit: Rolling Upgrades

TODO useful for upgrades (maybe in Discussion)

6.5.5 Dangerous corner case: Backup node active, dedicated link goes down, and there are new/reconnecting clients

TODO this will lead to split-brain syndrome (maybe in Discussion)

TODO in general: dedicated link for HA pair communication is dangerous!

6.5.6 Single Level

This is different from what's described in the [Zguide](#) because the concept of client requests is missing here (field devices don't request anything from Roadster nodes). What can be done instead is periodically sending life signs from one node to the other through the field device by updating some designated memory block. This can actually be done by both the active and the passive node, which reduces code complexity.

The passive node will check the active node's life signs periodically as well. In case the life signs cease, it can give its vote to the COMM BSTAR actor. This would satisfy the second condition of the [Binary Star Pattern](#) for a failover to take place. The first condition would be the missing heartbeats which are normally transmitted through the network link.

TODO new actor: COMM BSTARVOTER (or maybe directly into CORE actor)

6.5.6.1 Caveats

Special attention needs to be paid when it comes to writing these life signs. A naïve developer might implement the COMM BSTARVOTER so it autonomously causes life signs to be written on the field device. This works as long as the failures only affect the hardware. But what if a software

error happens in the CORE or BSTAR actor? They'd crash or hang, while the BSTARVOTER happily sends out life signs, which it obviously shouldn't be doing at that moment.

A better implementation would have the BSTARVOTER poll the BSTAR via the CORE router whether it's still alive, and only send out a life sign in case it gets an answer. This way, the BSTAR and the CORE actor are being tested for responsiveness. We'll call the two messages being sent back and forth "DEAD?" and "ALIVE!".

6.5.6.2 Link failure between Roadster node and field device

TODO describe why this won't cause a failover and thus can't be handled, as mentioned above

6.5.6.3 Supporting different field devices

TODO: adapter

6.5.7 Multi Level

This kind of HA setup is closely related to the [Binary Star Pattern](#) described in [6, Chapter 4 - Reliable Request-Reply Patterns, High-Availability Pair (Binary Star Pattern)]. This means that the passive node would actually receive requests from clients in case the active node fails, which simplifies the implementation. These requests will count as votes to fulfill the second condition that has to be met for a failover to be initiated. This is illustrated in [Figure 6.3](#).

6.6 Persistence Synchronization

The persisted data and updates to it, handled by the STORAGE actor, need to bubble up and collected in the root node.

6.6.1 Aspects

There are multiple aspects involved in persistence synchronization:

Delta: How does one get the initial delta of updates since last synchronization?

Updates: Further updates, one-by-one. This is only needed in case the solution aims for real-time synchronization.

HA peer sync: How does the inactive HA peer get updated? Of course, this only matters when the supernode is HA pair.

6.6.2 Variants

There are multiple variants to achieve the needed functionality.

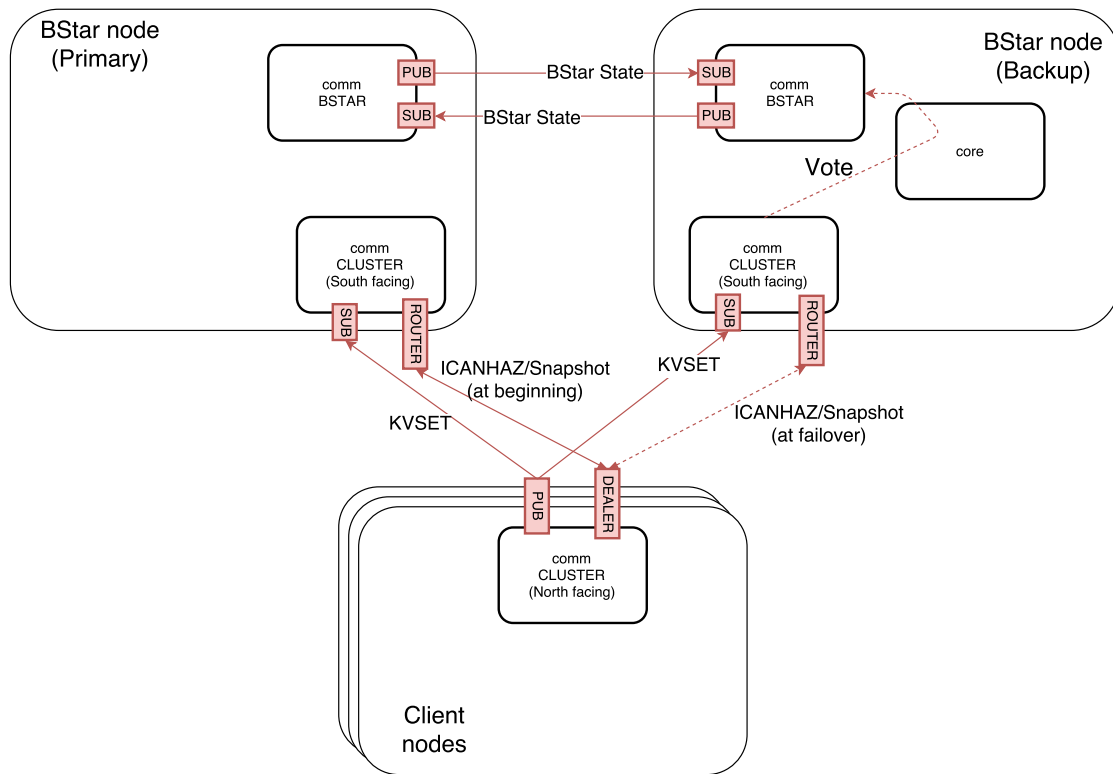


Figure 6.3: Multi level HA setup between a HA pair and a number of client nodes

6.6.2.1 Polling only

The supernode just periodically request persistence deltas. This would be handled over a DEALER/ROUTER pair of sockets. The nice thing about this variant is that the subnode only has to do one thing, which is responding to requests from the supernode(s); it doesn't have to proactively send any updates after sending the an initial delta.

A big drawback is that the synchronization doesn't happen in real-time. This doesn't seem to fit well into the overall Roadster architecture, which is completely event-driven (no polls or "sleeps").

Another drawback is efficiency. This variant will periodically cause the subnode's database to be searched for all keys. Depending on the size of the database and the efficiency of searching through keys, this could be a lot of wasted resources or even cause bottle necks when interacting with the STOR actor.

In case the supernode is a HA pair, this variant would generate duplicated traffic. To avoid this, another pair of sockets has to be introduced to synchronize persistence between a HA pair. This also means designing another protocol, and more moving parts overall.

Overall, this variant is very simple, but doesn't offer some features we'd normally expect from a framework like Roadster. The fruits are hanging low; achieving real-time synchronization and better efficiency is easy.

6.6.2.2 PUSH-PULL

This variant avoids the delays introduced by the polling mechanism of the first variant.

Procedure (for each subnode):

1. via a ROUTER/DEALER socket pair:
 - (a) supernode tells subnode its most recent timestamp in an ICANHAZ request
 - (b) subnode sends delta
 - (c) supernode receives and processes the complete delta
2. subnode sends updates to supernode via PUSH-PULL
3. during low-traffic times, we can send HUGZ as heartbeats

This seems nice at first, but PUSH socket's send buffer will fill up when the connection is interrupted. This isn't bad in and of itself, because when it's full (and writes start to block), we can just destroy the socket and reinitialize and start syncing anew (from ICANHAZ) after a certain timeout. But the problem is that, in case the delta is large, it will inevitably fill the PUSH socket's send buffer, temporarily reaching its high water mark, which is part of its normal operation.

So we'd have to introduce logic to recognize whether the PUSH socket is just temporarily full (e.g. during delta transmission), or permanently full (e.g. the supernode or the link to it is down).

Another disadvantage is that there needs to be another channel to synchronize persistence updates to the other HA peer, if there is one. This means another pair of sockets, another protocol to be designed, and more moving parts overall.

6.6.2.3 PUB/SUB

This is similar to [CSP/CHP](#). It's not 100% reliable, but even with unstable links, no data loss will occur if the client (the supernode) is able to reconnect within a specific amount of time. ØMQ's default for that amount is 10 seconds. As the requirements specify, 100% consistency is not mandatory for the persistent data.

A possible drawback is that the traffic is duplicated in case the supernode is a HA pair. However, there are numerous opportunities to mitigate this.

Procedure (for each subnode):

1. supernode subscribes to updates from subnode
2. via a ROUTER/DEALER socket pair:
 - (a) supernode tells subnode its most recent timestamp in an ICANHAZ request
 - (b) subnode sends delta
 - (c) supernode receives and processes the complete delta
3. supernode starts reading updates, possibly skipping the first few (based on timestamp)

6.6.3 Chosen Variant

We'll most likely go with the PUB-SUB variant, since it's simple and is similar to what's used for the new [CSP](#) in conjunction with multi-node [HA](#). It provides the best opportunities to improve

efficiency later on.

Its possible performance issues can be ignored right now, as trying to fix them is arguably considered premature optimization. If this turns out to be an issue in a productive deployment, like over a cellular network link, a future version can switch to multicast. ØMQ supports PGM, which is a reliable multicast protocol. (Pragmatic General Multicast, standardized, directly on top of IP, requires access to raw sockets and thus may require additional privileges) and EPGM (Encapsulated Pragmatic General Multicast, encapsulated in a series of UDP datagrams, doesn't require additional privileges, useful in a ØMQ-only setup).

If its reliability turn out to be an issue, one the socket option `ZMQ_RECOVERY_IVL` can be increased from 10 seconds to, say, 60 seconds, which gives an unstable link more time to recover before any data loss happens.

TODO: describe reasonable default setting, in case we change ZMQ's default.

6.7 Security

Encrypted communication can be achieved using ØMQ's transport encryption, which is completely transparent to the application. It uses the [libsodium](#) library to do this, although there's also the possibility to do it using [TweetNaCl](#) to avoid the additional dependency. The handshake is designed to be immune against flooding, using encrypted and authenticated cookies, so the server does not have to allocate any resources before the handshake is completed.

Server authentication is always done, by specifying the server's public key on the client side socket. Client authentication is optional. If done, the server side socket will talk to an additional socket responsible to authenticate clients by their public keys. The protocol between these two sockets is called [ZMQ Authentication Protocol \(ZAP\)](#). Completely abstracting the authentication in another socket allows any kind of authentication service to be plugged in.

6.8 OPC UA Interface: High Availability

TODO This is the optional goal.

TODO explain new opportunity for OPC UA HA server

TODO describe whatever needs to be described

[4, 6.4.2.4 Non-transparent Redundancy, p. 96] describes the exact behavior.

- study standard
- use Andy's gem
- according to Andy, this should be a simple thing

Chapter 7

Results

TODO what are the results (without discussing them)

TODO test results to verify our approach and implementations go here too

7.1 Port

TODO explain results here

7.2 Cluster

TODO explain results here

7.3 High Availability

TODO explain results here

Single Level HA

TODO explain results here

Multi Level HA

TODO explain results here

7.4 Persistence Synchronization

TODO explain results here

7.5 Security

TODO explain results here

7.6 OPC UA Interface: High Availability

TODO explain results here

Chapter 8

Discussion

TODO something like a SWOT analysis here (strengths, weaknesses, opportunities, threats)
TODO general advice: be concise, brief, and specific

8.1 Value Added

TODO what's better than before

8.2 Limitations

TODO identify potential limitations and weaknesses of the product

BStar pair has to be complete (both nodes running) during initialization. Otherwise, only primary node can serve requests; the backup node can't.

Message traffic towards root node sums up because of persistence synchronization. This shouldn't be a problem because of ØMQ's brilliant message batching, so the real limit is given by the inter-node network links.

8.3 Business Benefits

TODO potential applications (UeLS powered by Roadster?)

8.4 Ideas for Improvement

- HA within a node: kill and respawn an actor when it's unresponsive

- switch to Moneta for a unified key-value store interface, then eventually away from TokyoCabinet to something more modern and maintained, like LMDB (it's super fast and crash-proof)
- TIPC: high performance cluster communication protocol, suitable because Roadster nodes are Linux and there are direct links to peers (required for TIPC)
- client authentication
- key management in a DB (instead of files), with GUI to accept new clients
- dynamic node topology (maybe via DSL-file in Etcd, or DIM-only, or Zookeeper)
- other method for data serialization (like MessagePack), would allow adding other programming languages to the cluster
- fast compression for messages, like LZ4 or Snappy
- SERVER/CLIENT sockets from ZMQ 4.2 for simplified message routing

Chapter 9

Conclusion

TODO write conclusion, overall experience and opinion of product
TODO they should teach the actor model in APF, because ...

Bibliography

- [1] B. Batsov. *Ruby Style Guide*. URL: <https://github.com/bbatsov/ruby-style-guide> (visited on 10/06/2016).
- [2] *CAP theorem*. URL: https://en.wikipedia.org/wiki/CAP_theorem (visited on 10/10/2016).
- [3] A. Dworak, F. Ehm, P. Charrue, and W. Sliwinski. „The new CERN Controls Middleware“. In: *Journal of Physics: Conference Series* 396.012017 (2012). URL: <http://iopscience.iop.org/article/10.1088/1742-6596/396/1/012017/pdf> (visited on 10/06/2016).
- [4] OPC Foundation. *OPC Unified Architecture*. Part 4: Services. July 2015. URL: <https://opcfoundation.org/developer-tools/specifications-unified-architecture/part-4-services/> (visited on 10/06/2016).
- [5] Arnon Rotem-Gal-Oz. *Fallacies of Distributed Computing Explained*. URL: <http://www.rgoarchitects.com/Files/fallacies.pdf> (visited on 10/10/2016).
- [6] *Zguide*. URL: <http://zguide.zeromq.org/page:all> (visited on 10/11/2016).

Glossary

ØMQ High-performance socket library and concurrency framework for advanced messaging. [7](#), [8](#), [41](#), [42](#)

ACP Application Control Protocol. [10](#)

Actor Model A mathematical model for concurrent computation where there's no shared state and all communication between actors happens through messages. [7](#), [8](#), [53](#)

Binary Star Pattern A fairly simple hot-standby and failover mechanism to achieve high availability between two servers, described as a reliable request-reply pattern in the [Zguide](#). [28–31](#)

BSD Berkeley Software Distribution. [53](#)

C A compiled, imperative, very influential low-level programming language, invented in the early 1970s as a Unix system programming language. Compared to other languages, it very simple, knows only a handful of primitives and keywords.. [7](#)

case An alarm in a Roadster application that needs to be confirmed. [12](#)

CHP Clustered Hashmap Protocol. [18](#), [25](#), [33](#)

CI continuous integration. [21](#), [22](#)

Clone Pattern A client-server protocol to share state (a list of key-value pairs) across multiple clients, described as a reliable pub-sub pattern in the [Zguide](#). [12](#)

CSP Clone State Protocol. [10](#), [12](#), [16](#), [33](#)

CZMQ A thin abstraction layer ([wrapper façade](#)) for [ØMQ](#) with some additional functionality, written in clean and elegant C. [7](#), [23](#), [41](#), [54](#)

CZTop A modern, [Foreign Function Interface \(FFI\)](#) based [Ruby](#) binding for [CZMQ](#), written by Patrik Wenger. [7](#), [8](#), [23](#)

DIM Domain Information Model. [10](#), [12](#), [16](#), [18](#)

DSL Domain Specific Language. [26](#)

FFI Foreign Function Interface. [41](#)

HA high availability. [14](#), [16](#), [17](#), [19](#), [31](#), [33](#)

IoT Internet of Things. [9](#)

KISS The design principle “Keep it simple, stupid”, which favors simplicity over complexity. [20](#)

libsodium A portable and installable variant of [NaCl](#). <https://libsodium.org>. [7](#), [34](#)

LOG protocol Used within Roadster for system logging. [12](#)

MOM Message Oriented Middleware. [53](#)

NaCl Networking and Cryptography Library. Modern, state-of-the-art cryptography library, created by the Daniel J. Bernstein. <https://nacl.cr.yp.to>. 7, 41, 53

OPC Open Platform Communications. 9, 13

PCP Peer Control Protocol. 12

PDP Persistent Data Protocol. 12

PGM Pragmatic General Multicast. 53

PLC Programmable Logic Controller. 9, 10, 17, 22

RAID redundant array of independent disks. 9

RMP Roadster Messaging Protocols. 10

Ruby A modern and expressive scripting language from Japan. 7, 8, 41

RUP Rational Unified Process. 2

SCADA Supervisory Control and Data Acquisition. 8

SMP Supress Management Protocol. 12

SOAP Service Oriented Application Protocol. 9

SSD Solid State Disk. 9

TCP Transmission Control Protocol. 53

TDD test-driven development. 22

TIPC Transparent Inter-Process Communication. 53

TweetNaCl A compact, portable reimplementation of the NaCl in the form of 100 tweets, suited to be included it into one's trusted code base (as opposed to an external dependency). Implemented Daniel J. Bernstein et al. <http://tweetnacl.cr.yp.to>. 34

UA Unified Architecture. 9, 13

UI user interface. 10

Unix Domain Sockets Named pipes for extremely performant, duplex inter-process communication on Unix systems. 53

VM virtual machine. 21

wrapper façade A structural software design pattern which provides an object-oriented façade to a low-level functional subsystem or library. 41

ZAP ZMQ Authentication Protocol. 34

Zguide An extensive online document¹ describing best-practice patterns for ØMQ. 9, 12, 25, 28, 30, 41, 53

¹<http://zguide.zeromq.org/page:all>

Part III

Appendix

Appendix A

Self Reflection

TODO how did we perform, completion of goals, accuracy of estimated efforts, efficiency, resourcefulness

Appendix B

Task Description

The following five pages are the original task description, signed by Prof. Dr. F. Mehta.

Bachelor Thesis: Extending a SCADA framework to support high availability

1 Client and Supervisor

Client: mindclue GmbH

Client Contact: Andy Rohr, andy.rohr@mindclue.ch

Supervisor: Prof. Dr. Farhad Mehta, HSR Rapperswil

2 Students

- Patrik Wenger, pwenger@hsr.ch
- Manuel Schuler, mschuler@hsr.ch

3 Setting

The company mindclue GmbH, located in Ziegelbrücke, develops SCADA¹ applications for controlling systems used in traffic systems, energy, and water supply. For that purpose, the company developed the *Roadster* framework, which provides the basis for project specific applications.

Roadster is implemented in Ruby and is architecturally based on the Actor model [1], which means that multiple parallel running, single-threaded processes (actors) are coupled via messaging (shared-nothing architecture). The messaging layer is based on ZeroMQ (ZMQ [2]) and has an asynchronous/non-blocking nature. Several different messaging patterns/messaging protocols are used. Additionally, the system includes a web UI which is based on ember.js and connected to the messaging via WebSocket. Fundamentally, the system follows the Reactive Manifesto [3].

4 Goals

The main aim of this thesis is to extend the *Roadster* framework to support high availability.

Roadster currently lacks the following features:

1. A *Roadster* application is currently limited to one node (one instance). The goal is to be able to build systems which consist of multiple nodes. Example: A master node forms together with multiple subordinate nodes a system (basically a distributed system). The subordinate nodes are responsible for their respective subtask of a facility and communicate with their components (e.g. PLCs). The subsystems are integrated into the master node to form an overall view of the facility, which is visualized in the web UI.

This requirement implies:

¹Supervisory Control And Data Acquisition, see <https://en.wikipedia.org/wiki/SCADA>

- Extension of the messaging protocols to allow the communication between nodes across levels in the hierarchy.
 - Encryption of the communication.
2. A *Roadster* application has to have the ability to be run as a highly available active/passive cluster. Two nodes (primary and backup) at the same level in the hierarchy form a hot-standby cluster, where the two nodes stay in constant connection with each other. In case the active node fails, the passive node immediately takes over and becomes the new active node.

This requirement implies:

- Extension of the messaging protocols to allow the communication between nodes within the same level in the hierarchy.
 - Implementation of resilient failover mechanisms.
 - Encryption of the communication.
3. With (2), it is possible to implement a highly available OPC UA server. OPC UA [4] includes a concept for redundant UA servers. *Roadster* already implements a OPC UA server, although not highly available.

This requirement implies:

- Extension of the OPC UA implementation to support OPC UA HA mechanisms.

The client essentially wants *Roadster* to be extended by the three features described above, whereas the **third one is optional** and is only to be approached in case there is time for it. The **same applies to the encrypted communication requirement**.

5 Tasks

Here is an overview of the currently planned tasks that need to be performed:

1. Getting familiar with the concepts and implementation of *Roadster*, particularly the messaging layer. For that, Andy Rohr (mindclue GmbH) will provide an extensive introduction.
2. Elaboration of a subnode concept. This includes the design, implementation, and testing of extensions of the existing messaging protocols for the communication between nodes of different levels in the hierarchy.

One of the most important *Roadster* messaging protocols is called *Clone State Protocol* and is based on the *Clone Pattern* described in the zguide [5]. It provides means to replicate the current state of the domain model into the different actors within an application, as those actors behave according to the following principle [6]:

“Don’t communicate by sharing state; share state by communicating.”

For the communication between nodes, the protocol has to be extended accordingly. The messages are basically Ruby objects serialized using `Marshal.dump` and transported over ZMQ sockets.

3. Elaboration of a HA concept. This includes the design, implementation, and testing of extensions of the existing messaging protocols for the communication between nodes within the same level in the hierarchy, including resilient failover mechanisms.

The *Binary Star Pattern* [7][8] forms the basis of the HA concept. However, the concept will have to be adapted to fit *Roadster*'s needs. Availability has to be ensured under the following scenarios:

- hardware or software failure of the primary node
- network failure

A more detailed definition will have to be worked out during the elaboration phase of the thesis.

4. Implementation of encrypted communication. The current implementation is based on ZMQ 3 and the Ruby binding ffi-rzmq [9]. However, encryption has been introduced in ZMQ 4, which isn't supported by ffi-rzmq. On top of that, ffi-rzmq is not being maintained anymore. A possible solution is CZTop [10], which is based on CZMQ [11] and authored by Patrik Wenger.

In case CZTop is used, it would have to be extended to allow the watching of ZMQ sockets by EventMachine [12], e.g. `EM.watch(socket_file_descriptor)`. *Roadster* uses EventMachine as a reactor implementation [13].

5. Extensions of the *Roadster* OPC UA server implementation to support HA mechanisms. This part of *Roadster* is a Ruby extension, which is implemented based on the Unified Automation C++ SDK [14]. The extension is written in C++ and uses rbplusplus [15].

6 License

To grant mindclue GmbH unrestricted usage of the student's contributions, the student's code changes and additions shall be protected under the ISC License [16], which is functionally equivalent to the MIT license and the Simplified BSD license, but uses simpler language.

7 Guidelines

The students and the supervisor will plan weekly meetings to check and discuss progress. The student will schedule meetings with the client as and when required (recommendation: 1 meeting per week of 1 hour duration).

All meetings are to be prepared by the students with an agenda. The agenda will be sent at least 24h prior to the meeting. The results will be documented in meeting minutes that will be sent to the supervisor.

A project plan must be developed at the beginning of the thesis to promote continuous and visible work progress. For every milestone defined in the project plan, the temporary versions of all artefacts need to be submitted. The students will receive a provisional feedback for the submitted milestone results. The definitive grading is however only based on the final results of the formally submitted report.

8 Documentation

The project must be documented according to the regulations of the Computer Science Department at HSR [17]. All required documents are to be listed in the project plan. All documents must be continuously updated, and should document the project results in a consistent form upon final submission. All documentation and work artefacts have to be completely submitted

in three copies on CD/DVD (one copy each for the client, university, and supervisor). Three printed copies of the report need to be submitted (one copy each for the client, external examiner, and supervisor).

9 Important Dates

Please refer to <https://www.hsr.ch/Termine-Diplom-Bachelor-und.5142.0.html>.

10 Workload

A successful Bachelor thesis project results in 12 ECTS credit points per student. One ECTS point corresponds to a work effort of 30 hours. All time spent on the project must be recorded and documented.

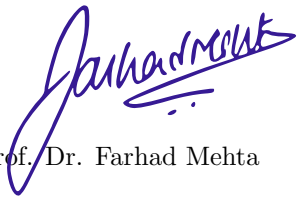
11 Grading

The HSR supervisor is responsible for grading the master thesis. The following table gives an overview of the weights used for grading.

Facet	Weight
1. Organisation, Execution	1/6
2. Report	1/6
3. Content	3/6
4. Final Presentation & Examination	1/6

The effective regulations of the HSR and Department of Computer Science [18] apply.

Rapperswil, Wednesday 28th September, 2016



Prof. Dr. Farhad Mehta

References

- [1] URL: https://en.wikipedia.org/wiki/Actor_model.
- [2] URL: <http://zeromq.org/>.
- [3] URL: <http://www.reactivemanifesto.org/>.
- [4] URL: <https://opcfoundation.org/about/opc-technologies/opc-ua/>.
- [5] URL: <http://zguide.zeromq.org/page:all#Reliable-Pub-Sub-Clone-Pattern>.
- [6] URL: <https://www.igvita.com/2010/12/02/concurrency-with-actors-goroutines-ruby/>.
- [7] URL: <http://zguide.zeromq.org/page:all#High-Availability-Pair-Binary-Star-Pattern>.
- [8] URL: <http://zguide.zeromq.org/page:all#Adding-the-Binary-Star-Pattern-for-Reliability>.
- [9] URL: <https://github.com/chuckremes/ffi-rmq>.
- [10] URL: <https://github.com/paddor/cztop>.
- [11] URL: <http://czmq.zeromq.org>.
- [12] URL: <http://www.rubydoc.info/gems/eventmachine>.
- [13] URL: https://en.wikipedia.org/wiki/Reactor_pattern.
- [14] URL: <https://www.unified-automation.com/products/server-sdk/c-ua-server-sdk.html>.
- [15] URL: <https://github.com/jasonroelofs/rbplusplus>.
- [16] URL: https://en.wikipedia.org/wiki/ISC_license.
- [17] URL: <https://www.hsr.ch/Allgemeine-Infos-Bachelor-und.4418.0.html>.
- [18] URL: <https://www.hsr.ch/Ablaeufe-und-Regelungen-Studie.7479.0.html>.

Appendix C

License

As stated in the task description, all of our code contributions underlie the ISC license, which is functionally equivalent to the MIT license and the Simplified BSD license, but uses simpler language. In addition to that, we hereby explicitly grant mindclue GmbH unrestricted usage of all our code contributions.

Appendix D

Project Plan

TODO import project plan from wiki
TODO import risks from wiki

Appendix E

ØMQ

ØMQ is a [Message Oriented Middleware \(MOM\)](#) implemented as an open source library, that is, it doesn't require a dedicated broker. Instead, it offers sockets with an abstract interface similar to [BSD](#) sockets. Different types of sockets are used for different messaging patterns such as request-reply, publish-subscribe, and push-pull.

A single socket can bind/connect to multiple endpoints, which allows ØMQ to use round-robin on the sender side, and fair-queueing on the receiver side, where applicable. It doesn't matter whether the communication happens in-process (between threads), inter-process (e.g. over [Unix Domain Socketss](#)), or inter-node (e.g. over [TCP/PGM/TIPC](#)), since the transport is completely abstracted away. The same goes for connection handling; an arbitrary amount of connections is handled over a single socket and reconnecting after short network failures is done transparently.

ØMQ is lightweight and allows for extremely low latencies, which means it can also be used as the fabric of concurrent applications, e.g. for the [Actor Model](#). In case of the TCP transport, it incorporates advanced techniques such as smart message batching to achieve significantly higher throughputs than with raw TCP or other [MOM](#) solutions [3, Figure 2, Middleware evaluation and prototyping, p. 4].

To build a solution with ØMQ, its sockets are used as building blocks to design custom message flows. Certain patterns are used to achieve reliability with respect to the failure types that need to be addressed in particular. The [Zguide](#) explains best practices, including commonly needed, resilient messaging patterns.

The above characteristics make ØMQ a valuable asset when it comes to building robust, distributed high-performance systems.

E.1 Transport Security

Since version 4.0, ØMQ boasts state of the art encryption and authentication, based on the excellent and highly renown [NaCl](#)¹ library.

¹<http://nacl.cr.yp.to>

E.2 Data Serialization

Data serialization is outside the scope of ØMQ. To fill the gap, one typically uses another library such as MsgPack², Protocol Buffers³, or even a programming language's built-in object serialization support⁴.

E.3 CZMQ

CZMQ is a high-level abstraction layer for ØMQ. It makes working with the ØMQ library more expressive and allows for better portability. It also provides additional functionality such as a reactor, a simple actor implementation, as well as utilities for certificate and authentication handling, and LAN node discovery. This is the recommended way of using ØMQ nowadays, as it allows for much cleaner C code and also simplifies bindings for other languages.

²<http://msgpack.org>

³<https://developers.google.com/protocol-buffers/>

⁴such as Ruby's marshalling support: <http://ruby-doc.org/core/Marshal.html>

Appendix F

Infrastructural Problems

TODO describe serious problems here, if any

F.1 Project Management Software

TODO Github/Trello/Harvest/Everhour/Elegantt/Ganttify/Redmine