

BACHELOR THESIS

Extension of a SCADA Framework to support High Availability and Security

#Ruby #ØMQ

Patrik Wenger, Manuel Schuler

client: mindclue GmbH

supervisor: Prof. Dr. Farhad Mehta

expert: Prof. John Doe

September – December, 2016

Abstract

TODO introduction
TODO approach and technologies
TODO result

Declaration of Originality

We hereby confirm that we are the sole authors of this document and the described changes to the Roadster framework and libraries developed.

TODO any usage agreements or license

Acknowledgements

TODO anyone we'd like to thank

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	2
1.5	Personal Goals	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	ØMQ	8
4.2.3	Software Architecture	9
4.3	Goals	12
5	Requirements	13
5.1	Functional	13
5.1.1	Cluster	13
5.1.2	Single Level HA	14
5.1.3	Multi Level HA	14
5.1.4	Persistence Synchronization	14
5.1.5	OPC UA HA	15
5.2	Use Cases	15
5.2.1	UC01: Cluster	15
5.2.2	UC02: Hardware failure at top	15
5.2.3	UC03: Hardware failure at bottom	15
5.2.4	UC04: Persistence synchronization	15
5.2.5	UC05: OPC-UA HA	15
5.3	Non-Functional Requirements	15
5.3.1	Simplicity	15
5.3.2	Testing	15

5.3.3	Security	16
5.3.4	Coding Guidelines	16
6	Approach	17
6.1	Getting Familiar with Roadster	17
6.2	Testing	17
6.2.1	Setup	17
6.2.2	Unit test	17
6.2.3	Integration test	18
6.2.4	Continuous integration	18
6.2.5	System test	18
6.3	Port to new ØMQ library	19
6.4	Cluster	19
6.4.1	Aspects	20
6.4.2	DIM Synchronization	20
6.4.3	Node Typology Definition	21
6.4.4	Message Routing	23
6.5	High Availability	23
6.5.1	Defining Reliability	24
6.5.2	What's Needed	24
6.5.3	Binary Star in a Nutshell	24
6.5.4	Failover	24
6.5.5	Side benefit: Rolling Upgrades	25
6.5.6	Dangerous corner case: Backup node active, dedicated link goes down, and there are new/reconnecting clients	25
6.5.7	Single Level	25
6.5.8	Multi Level	26
6.6	Persistence Synchronization	27
6.6.1	Aspects	27
6.6.2	Variants	27
6.6.3	Chosen Variant	28
6.7	Security	28
6.8	OPC UA Interface: High Availability	29
7	Results	30
7.1	Port	30
7.2	Cluster	30
7.3	High Availability	30
7.4	Persistence Synchronization	30
7.5	Security	30
7.6	OPC UA Interface: High Availability	31
8	Discussion	32
8.1	Value Added	32
8.2	Limitations	32
8.3	Business Benefits	32
8.4	Ideas for Improvement	32
9	Conclusion	34
	Glossary	36
III	Appendix	38
A	Self Reflection	39

B Task Description	40
C License	46
D Project Plan	47
E ØMQ	48
F Infrastructural Problems	49
F.1 Project Management Software	49

List of Figures

4.1	Roadster's software architecture	10
4.2	Roadster's communication layers	11
6.1	Cluster setup between a supernode and two subnodes	22
6.2	Single level HA setup between a HA pair and a PLC	25
6.3	Multi level HA setup between a HA pair and a number of client nodes	26

List of Tables

Listings

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.

1.5 Personal Goals

Next to excelling in this bachelor thesis, the aim is also 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.

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

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](#)

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, Roadster is a next-generation SCADA framework, and the results of this thesis 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. 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

TODO spread of CZTop and proof of its design

TODO creation of cztop-patterns (as mentioned in the Mgmt Summary)

4.2 Initial Situation

4.2.1 mindclue GmbH

About

The company mindclue GmbH, located in Ziegelbrücke provides its partner REMTEC AG with complete SCADA applications. These are then used to control and monitor operational 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.

Roadster

Roadster ist eine Ereignis gesteuerte Applikation geschrieben in Ruby. Sie ist die eigentliche [Supervisory Control and Data Acquisition \(SCADA\)](#) Applikation und wird bereits in einer Vielzahl von Tunnelanlagen in der Schweiz eingesetzt und dient zur Überwachung der einzelnen Komponenten im Tunnel. Durch den modularen Aufbau kann jede Roadster Applikation sich selbst laufen - Autonom. Sprich jeder Roadster hat seinen eigenen Webserver und Datenverwaltung.

AS - UeLS

Roadster ist einer von vielen AS Knoten eines UeLS. Die Kommunikation zwischen AS und AR geschieht über "OPC UA".

4.2.2 ØMQ

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

To understand Roadster’s architecture and the rest of this document, it’s helpful to understand the basics of ØMQ (sometimes written as ZeroMQ or simply ZMQ) first. This is a brief introduction to ØMQ for the unfamiliar reader.

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 Sockets](#)), 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 provides 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 [2, 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](#)³ 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.

Transport Security

Since version 4.0, ØMQ boasts state of the art encryption and authentication, based on the excellent and highly renown [NaCl](#)⁴ library.

Data Serialization

Data serialization is outside the scope of ØMQ. To fill the gap, one typically uses another library such as [MsgPack](#)⁵, [Protocol Buffers](#)⁶, or even a programming language’s built-in object serialization support⁷.

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.

4.2.3 Software Architecture

TODO more

Roadster is event-driven 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:

³<http://zguide.zeromq.org/>

⁴<http://nacl.cr.yp.to>

⁵<http://msgpack.org>

⁶<https://developers.google.com/protocol-buffers/>

⁷such as Ruby’s marshalling support: <http://ruby-doc.org/core/Marshal.html>

“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. This can be different kind of [Programmable Logic Controllers \(PLCs\)](#) or higher level monitoring systems. Each COMM actor uses an adapter specifically written for a single communication protocol.

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.

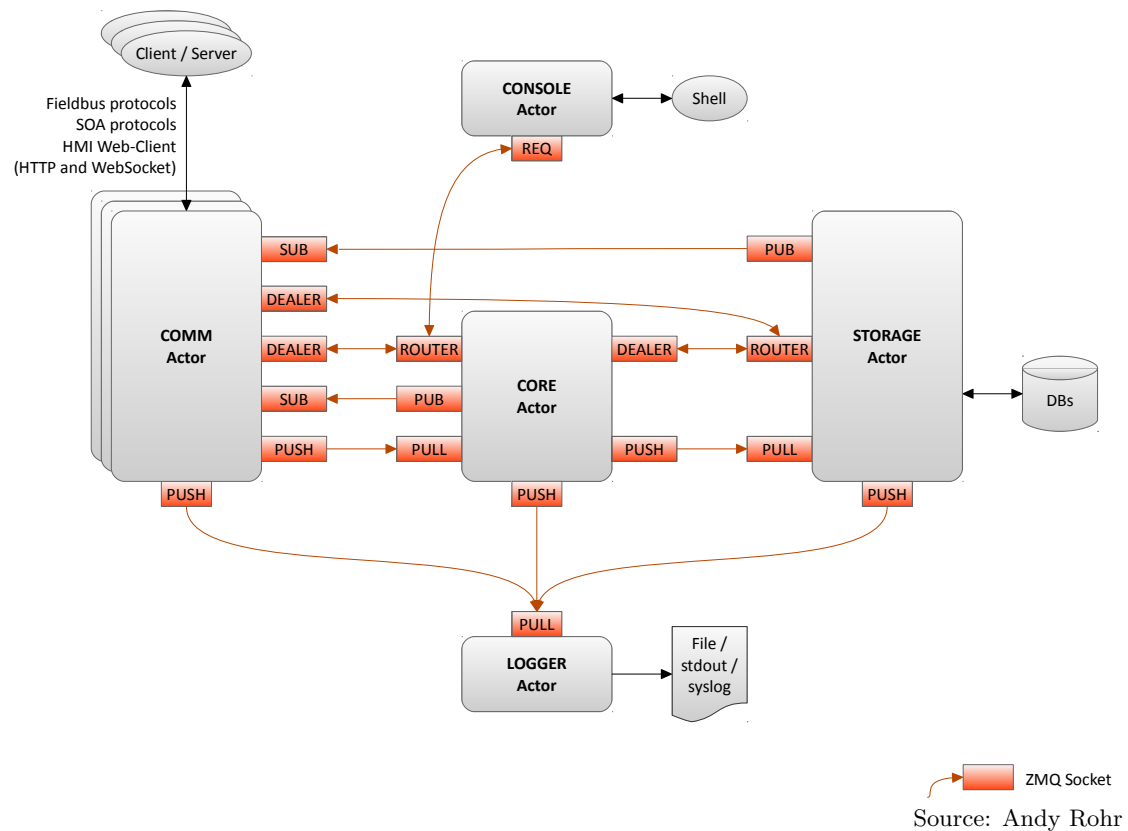


Figure 4.1: Roadster’s software architecture

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:

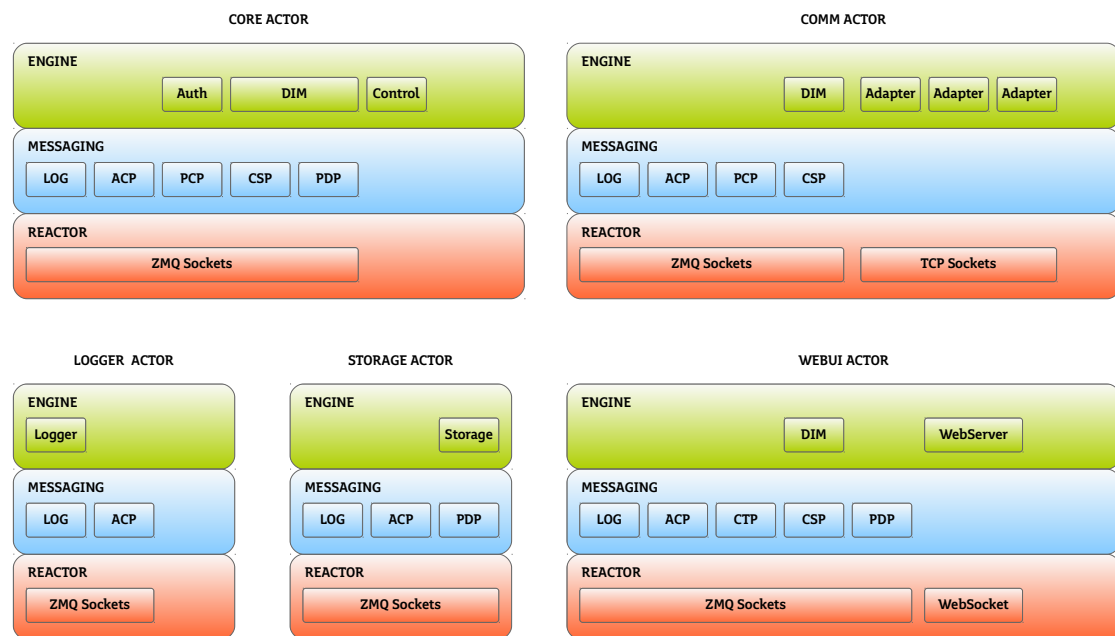
Here is the business logic of Roadster, e.g. the [Domain Information Model \(DIM\)](#), user authentication, adapters for different devices, the web [user interface \(UI\)](#), etc.

Messaging layer:

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

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](#).



Source: Andy Rohr

Figure 4.2: Roadster's communication layers

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.

[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.

[Command Transaction Protocol \(CTP\):](#)

Used for asynchronous command execution with feedback.

LOG protocol:

Used for system logging.

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

4.3 Goals

TODO preprocessed mandatory goals

TODO retrospective, diff with initial goals

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

1. Extending communication protocols to support clustering
2. Extending communication protocols to add high availability

An implication of the above goals is secure inter-node communication in a Roadster cluster. This is also to mitigate common security concerns with SCADA systems which are becoming more and more open due to standardization. To quote wikipedia:

TODO move this part somewhere else, a section dedicated to security in SCADA systems

“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.”

Optional Goals

TODO optional goals

1. Encryption of the communication
2. Providing of the highly available [Open Platform Communications \(OPC\) Unified Architecture \(UA\)](#) server interface

TODO The Eight Fallacies of Distributed Computing <https://blogs.oracle.com/jag/resource/Fallacies.html>

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

Below are the functional requirements, as opposed to the non-functional requirements.

5.1.1 Cluster

This requirement is to allow running Roadster on multiple nodes in a hierarchical topology.

Extending the [CSP](#) to keep the [DIM](#) in sync across all nodes is a central part of the clustering functionality. This means synchronizing the items within the DIM that have `@lifecycle_status = "updated"`. 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](#)

In addition to that, there needs to be some kind of message routing 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.

It's important that every node subtree can live on autonomously even if the link to its supernode or the supernode itself fails.

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 for all other nodes in the setup.

Typical Usecases

The typical use cases include:

Single level, single node This is the legacy setup and is what Roadster is already able to do.

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.

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.

The exotic cases which can be ignored, if need be, include:

- Multi level, HA at bottom
- Multi level, HA in the middle

5.1.2 Single Level HA

This is where there's a node pair directly connected to a PLC. Both nodes have read/write access to the PLC, but only one of the nodes (the active one) must do so. The nodes must automatically find consensus on who's active. The passive one must automatically take over in case the active one is confirmed to be dead.

The kinds of failures that need to be handled include:

- Hardware/software failure on the primary node
- Failure of the link between the two nodes

5.1.3 Multi Level HA

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

The kinds of failures that need to be handled include:

- Hardware/software failure on the primary node
- Failure of the link between the two nodes

5.1.4 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.

5.1.5 OPC UA HA

This is optional.

A given HA pair needs to provide an OPC UA Server Redundancy interface, as described in [3, 6.4.2.4 Non-transparent Redundancy, p. 96].

5.2 Use Cases

The use cases are briefly described here.

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

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

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 ZMQ sockets.

The communication between a given set of Roadster nodes must be secured using encryption.

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

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

TODO: this is the first task according to Task Description
TODO andy's introduction during first Friday morning
TODO first impression of Roadster's code base: very clean, abstractions, loosely coupled, scarcely documented

6.2 Testing

Diese Section beschreibt die Testmethods welche wir angewandt haben, um die einzelnen Funktionen, das Zusammenspiel der Komponenten, sowie das Verhalten der Applikation als ganzes zu testen. Alle Resultate zu den Tests können im Results Kapitel entnommen werden.

This section describes the test methods we used to check single functions, the components interaction and the behaviour of the whole application. All results of the tests can be found in the results section.

6.2.1 Setup

Zur Umsetzung von Continuous Integration wird gitlab CI verwendet, weil gitlab kostenlos ist im gegensatz zu seinem pendant Travis CI, welches nach mehr als 100 Builds kostet bei privaten repositories. Daher wurde gitlab CI auf dem HSR Server installiert und konfiguriert.

Due to the fact that our repository is private, we can't use the online travis ci without to pay for it, because after 100 build it costs. We had to install on our hsr server gitlab. Gitlab has a own continuous integration which we used for unit and integration test. Each commit execute all unit tests and integration tests. This is straightforward to check if all components work together correctly.

6.2.2 Unit test

Um die Richtigkeit der Implementationen zu gewährleisten wurden Unittests mit Ruby Unit gesetzt, welche die Kernfunktionen testen. Die Unit Tests sind im Source Code zu finden.

To ensure the correctness of the implementations were ruby unit tests set which test the functions. Whenever the functionality of the application was changed or a bug was fixed, new tests

were added to cover the new behavior. An other benefit of the tests was, we could refactoring the code and check if nothing on the behaviour changed. For each commit to the repository every test has to pass otherwise it is an offend against our team contract.

The unit tests can be found in the source code.

6.2.3 Integration test

Integration tests überprüfen das Zusammenspiel zwischen den einzelnen Komponenten. Um unsere Kern Features zu testen (Cluster, HA, persistence) schreiben wir integration tests mit ruby unit tests. Dazu starten wir jeden Node als eigenen Prozess. ZMQ erlaubt es uns die "Inter-Kommunikation" auch über Prozesse zu testen. Um Verbindungsprobleme zu testen werden die einzelnen Prozess "gekillt" um ein Failover zu erzwingen.

Integration tests verify the interaction between the individual components. To test our core features (like cluster, high availability, persistence). We wrote integration tests with ruby unit tests. To test the interaction between nodes, we started the roadster application in different process. ZMQ allows us to interact between process and they will act like internode connections. In order to test the failover or synchronization functionality the individual process - "node" can be killed and started again later if the scenario defines this.

More detail about the integration tests scenarios can be found under the results chapter.

6.2.4 Continous integration

Continous Integration soll Aushilfe schaffen beim Eliminieren von Integration Problemen. Daher wird CI bei jedem Push ins Github Repository ausgeführt. Dazu überwacht gitlab das Repository und führt das definierte Build zum Projekt aus Dieses Build beinhaltet das installieren von Abhängigkeiten, das Installieren der Roadster app und zu guter letzt das Ausführen der Tests.

Continous integration helps us to prevent integration problems also known as "integration hell". On each push to the repository will trigger an CI check. Therefore gitlab hooks our repository and runs the defined build from the project. A Build contains the installation of the dependencies, installing the roadster app and the carrying out of tests.

6.2.5 System test

Systemtests dienen dazu die Applikation unter realen Bediengungen 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 Netzkapazitätprobleme 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 VirtualMachine'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.

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

TODO justify why port is needed right at the beginning (exclude faults from unmaintained ffi-rmq gem, encryption is needed anyway, all the other tasks involve communication over ZMQ)

TODO explain binding options out there, why CZTop (including difference between ZMQ and CZMQ)

TODO explain preliminary task of adding support for the ZMQ options ZMQ_FD and ZMQ_EVENTS in CZTop

TODO explain concept of exchanging ffi-rmq with CZTop

6.4 Cluster

TODO describe scribble (chp.pdf)

6.4.1 Aspects

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?

6.4.2 DIM Synchronization

TODO election/design of appropriate protocol

The Existing CSP in a Nutshell

This is a brief introduction/refreshers for the Clone State Pattern implemented by Roadster. 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 messages 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.

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.

What's Missing

TODO it has to work across several nodes

TODO it has to be able to handle HA supernodes

TODO explain Clustered Hashmap Protocol (?)

TODO PCP: use DIM to know node tree and determine next hop for (dialog or fire+forget) messages TODO mention CAP theorem, we need AP since sub trees are autonomous, but we guarantee eventual consistency by restricting write access to the owning node

Here are the possible variants:

Variant 1. Sync self-subtree only:

- always sync on self-subtree only
- con: no copy of remaining tree

Variant 2. Sync complete tree:

- always sync on complete tree
- get snapshot and merge own subtree

Variant 3. Either sync on subtree or complete tree:

- make it configurable: either sync on subtree or complete tree

Variant 2 will be the first step. Variant 3 will be the second step, if at all.

What's needed

Since COMM actors are used to communicate with the outside world, new COMM actors will have to be introduced: One kind that is south-facing to communicate with subnodes, and another one that is north-facing for communication with supernodes. We'll call them COMM CLUSTER NORTH and COMM CLUSTER SOUTH.

Their responsibility is the inter-node synchronization of the DIM, similarly to what's happening in the existing CSP within a single node.

- they do something closely related to the existing CSP
- future oriented: because of the HA requirement, ideas from the CHP are integrated, such as using PUB-SUB (instead of PUSH-PULL) for inter-node KVSET messages, so both super nodes (in HA setup) hear updates
- for intra-node KVSET messages, PUSH-PULL is OK and can be left unchanged
- only node A has write access to values on A (to avoid uncertain situations involving race conditions), e.g.:

6.4.3 Node Typology Definition

- node topology in DSL, static file (e.g. topology.conf.rb) shared on all nodes, read by each actor on startup
- specific config file on each node (conf.rb) knows its own place in topology (through `conf.system_id`)
- maybe a HA pair is one DIM object, has one name, but two IP addresses (primary and backup, in order)

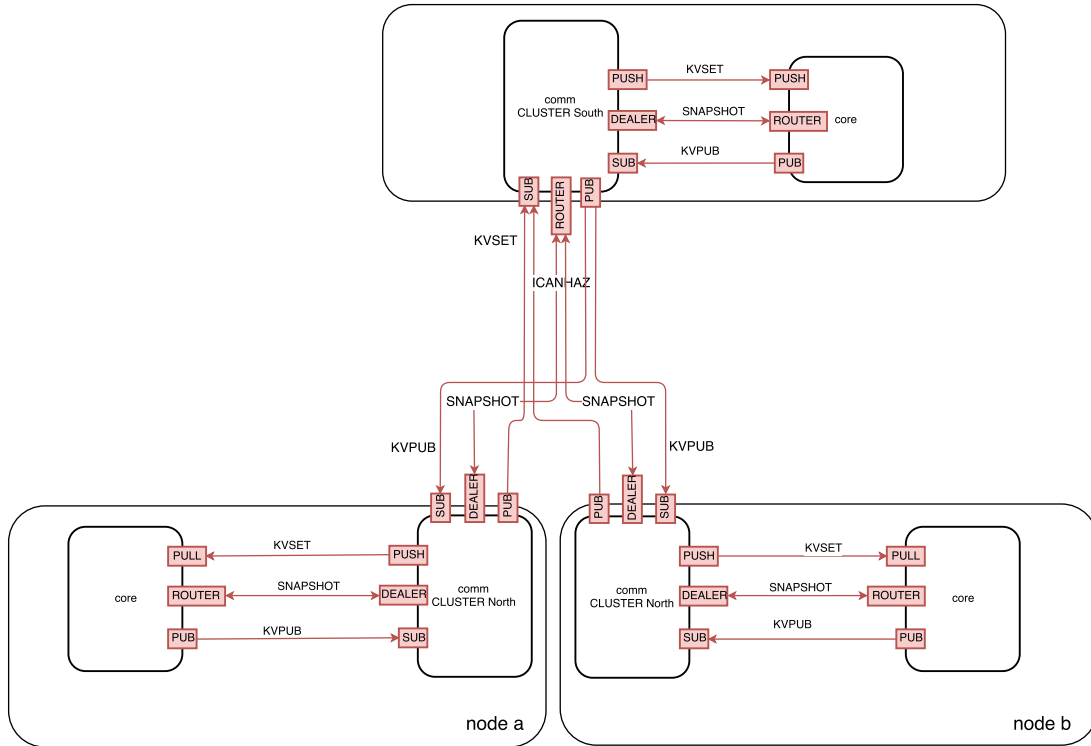


Figure 6.1: Cluster setup between a supernode and two subnodes

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

#####
# without HA:

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"

#####
# later with 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

# 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]"

#####
# within ba-roadster-app's lib/domain/domain.rb file:
#
# 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
end

end # Domain::Model
end # Roadster

```

6.4.4 Message Routing

TODO message routing (end-to-end routing with identity/identities as prepended message frame?, should be simpler and more efficient than hop-by-hop routing)

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 between the two HA peers. 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 the 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 PLC 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 PLC could cause an alarm, but no failover, since they're only half of the conditions that have to be met for a failover.

6.5.2 What's Needed

TODO A simple HA protocol =, Binary Star

TODO maybe we need to extend binary star to handle other kinds of network failures?

TODO new actor: COMM BSTAR

6.5.3 Binary Star in a Nutshell

TODO explain binary star

TODO two conditions (no heartbeats, and vote)

6.5.4 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 (ICANHAI 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.

Alarm Generation

When a failover happens, we want to create a [Case](#) (alarm) in the DIM, so the outage is visible to a user in one of the web UIs. Also, in case the passive node goes down, which doesn't have an immediate effect on availability, we also want to create a [Case](#). This is so the user can act upon the alarm and e.g. initiate field forces to inspect the failed node and repair it.

Once repaired, it's started with the exact same configuration (either primary or backup) again. 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.

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.

6.5.5 Side benefit: Rolling Upgrades

TODO useful for upgrades (maybe in Discussion)

6.5.6 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!!

6.5.7 Single Level

This is different from what's described in the zguide because the concept of client requests is missing here (PLCs don't request anything). What can be done instead is periodically sending life signs from one node to the other through the PLC 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 PLCVOTER

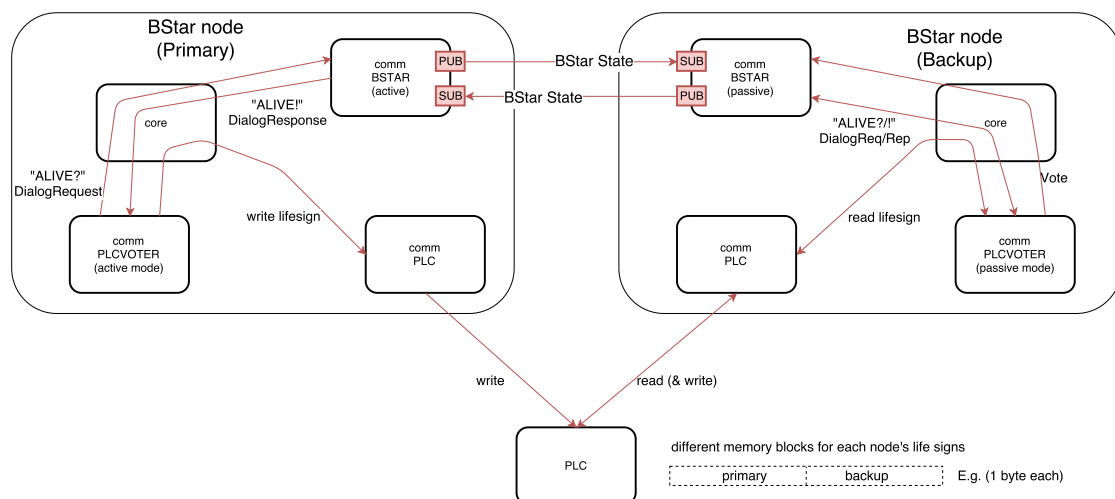


Figure 6.2: Single level HA setup between a HA pair and a PLC

Caveats

Special attention needs to be paid when it comes to writing these life signs. A naïve developer might implement the COMM PLCVOTER so it autonomously causes life signs to be written PLC. 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 PLCVOTER happily sends out life signs, which it obviously shouldn't be doing at that moment.

A better implementation would have the PLCVOTER 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!".

Node—PLC Link Failure

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

Supporting Different PLCs

TODO: adapter

6.5.8 Multi Level

This kind of HA setup is closely related to the [Binary Star Pattern](#) described in the [Zguide](#), meaning that there will be actual requests from client on the passive node in case the active node fails. These requests will count as votes to fulfill the second condition that has to be met for a failover to be initiated.

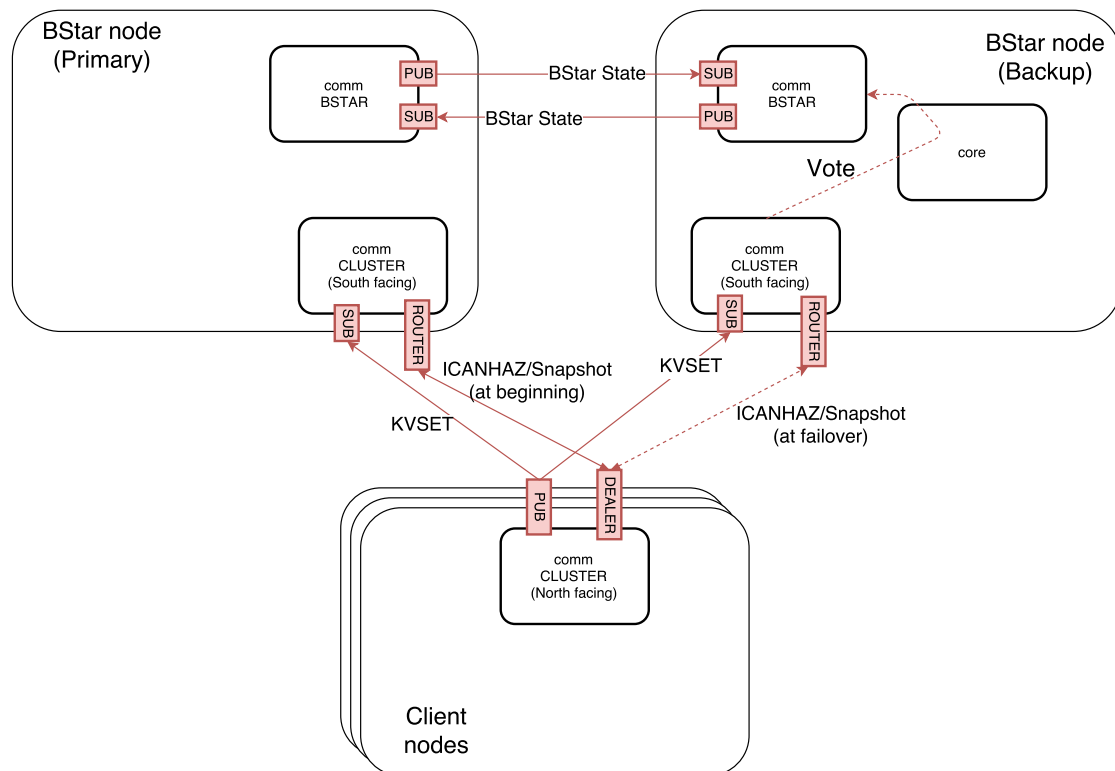


Figure 6.3: Multi level HA setup between a HA pair and a number of client nodes

6.6 Persistence Synchronization

The persisted data needs to be bubbled up towards 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.

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").

TODO another drawback: computing the diff might be heavy (depending on DB)

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 is easy.

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.

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

[3, 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] 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).
- [3] 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).

Glossary

ØMQ High-performance socket library and concurrency framework for advanced messaging. [7](#), [36](#), [37](#)

ACP Application Control Protocol. [11](#)

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

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](#). [24–26](#)

BSD Berkeley Software Distribution. [9](#)

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. [11](#)

CHP Clustered Hashmap Protocol. [14](#), [28](#)

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](#). [20](#)

CSP Clone State Protocol. [11](#), [13](#), [20](#), [28](#)

CTP Command Transaction Protocol. [11](#)

CZMQ A thin abstraction layer ([wrapper façade](#)) for [ØMQ](#) with some additional functionality, written in clean and elegant C. [7](#), [9](#), [36](#)

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

DIM Domain Information Model. [11](#), [13](#), [14](#)

FFI Foreign Function Interface. [36](#)

HA high availability. [13–15](#), [26](#), [28](#)

KISS The design principle “Keep it simple, stupid”, which favors simplicity over complexity. [15](#)

libsodium A portable and installable variant of [NaCl](#). <https://libsodium.org>. [7](#), [28](#)

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

MOM Message Oriented Middleware. [9](#)

NaCl Networking and Cryptography Library. Modern, state-of-the-art cryptography library, created by the Daniel J. Bernstein. <https://nacl.cr.yp.to>. [7](#), [9](#), [36](#)

OPC Open Platform Communications. [12](#)

PDP Persistent Data Protocol. [11](#)

PGM Pragmatic General Multicast. [9](#)

PLC Programmable Logic Controller. [10](#), [11](#), [18](#), [19](#)

RMP Roadster Messaging Protocols. [11](#)

Ruby A modern and expressive scripting language from Japan. [7](#), [36](#)

RUP Rational Unified Process. [2](#)

SCADA Supervisory Control and Data Acquisition. [8](#)

SMP Supress Management Protocol. [11](#)

TCP Transmission Control Protocol. [9](#)

TIPC Transparent Inter-Process Communication. [9](#)

TweetNaCl A compact, portable reimplementaion 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>. [28](#)

UA Unified Architecture. [12](#)

UI user interface. [11](#)

Unix Domain Sockets Named pipes for extremely performant, duplex inter-process communication on Unix systems. [9](#)

wrapper façade A structural software design pattern which provides an object-oriented façade to a low-level functional subsystem or library. [36](#)

ZAP ZMQ Authentication Protocol. [28](#)

Zguide An extensive online document¹ describing best-practice patterns for ØMQ. [20](#), [26](#), [36](#)

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.

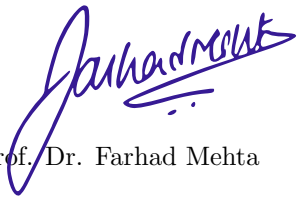
11 Grading

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

TODO explain ZMQ in greater detail

TODO strong abstraction (one socket for many connections, connection handling transparent, transport and encryption transparent, no concept of peer addresses)

TODO brokerless/with broker, up to you

TODO basic patterns

TODO extended patterns

TODO not only a "MOM", but a multi threading library (Actor pattern)

Appendix F

Infrastructural Problems

TODO describe serious problems here, if any

F.1 Project Management Software

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