HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

roadster
advanced web scada system

Bachelor thesis

Roadster High Availability

Patrik Wenger, Manuel Schuler

for industry client
mindclue GmbH

supervised by
Prof. Farhad Mehta

Fall semester 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

Acknoledgements

TODO anyone we'd like to thank

Management Summary

Context

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 availablity, and secure network communications.

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 architecure 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.

Software Development Process

TODO take RUP into glossary

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.

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.

Personal Goals

Next to excelling in this bachelor thesis, the students also hope to create a reusable open-source library as a by-product. The intention is that the library makes certain ØMQ-based communication protocols readily available for other developers facing the same problems.

Project Phases

TODO describe this phase in retrospection

Inception

TODO include Gantt chart for this phase
TODO describe this phase in retrospection

Elaboration

TODO include Gantt chart for this phase
TODO describe this phase in retrospection

Construction

TODO include Gantt chart for this phase
TODO describe this phase in retrospection

Transition

TODO include Gantt chart for this phase
TODO describe this phase in retrospection

Results

TODO describe results

Contents

List of Figures

List of Tables

Listings

Part I

Technical Report

Chapter 1

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.

Motivation

TODO Why do we care about this thesis? Why are we interested?

To better understand our motivation, it might help to understand our personal backgrounds first.
TODO: Patrik: Ruby, creation of CZTop, fascination with Actor model (e.g. Erlang, Celluloid in Ruby), interest in Pony, interest in distributed systems, huge interest in security
TODO: Manuel: Javascript, .NET, horizont extension, learning new things, no hesitation for this project

We're thrilled about the following technologies and software design patterns:

- ØMQ
- Ruby
- Actor Model
- Distributed Computing
- High Availability

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.

In addition to that, we look at this bachelor thesis as an opportunity to become more fluent in English, as well as a way 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 not, 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.

Open-Source Engagement

TODO spread of CZTop and proof of its design
TODO creation of cztop-patterns (as mentioned in the Mgmt Summary)

Initial Situation

mindclue GmbH

About

Die Firma mindclue GmbH mit Sitz in Ziegelbrücke GL/SG stellt für ihren Partner REMTEC AG für Steuerungen, Regelungen und Überwachungen von technischen Prozessen, komplette SCADA- und Steuerungssysteme her. Dabei sind sie in den Bereichen Betriebs-und Sicherheitausrüstung für Nationalstrassen, Trinkwasser- versorgungen, im Energiesektor und vielen Spezielbereichen tätig. Dabei setzen sie auf ihre eigens entwickelte Applikation - Roadster.

Roadster

Roadster ist eine Ereignis gesteuerte Applikation geschrieben in Ruby. Sie ist die eigentliche 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".

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 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-robbin 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[1] explains best practices, including commonly needed, resilient messaging patterns.

[1] `http://zguide.zeromq.org/`

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[2] library.

Data Serialization

Data serialization is outside the scope of ØMQ. To fill the gap, one typically uses another library such as MsgPack[3], Protocol Buffers[4], or even a programming language's built-in object serialization support[5].

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.

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:

> "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 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 1.1 illustrates Roadster's architecture.

Communication Layers

TODO briefly explain layers. Figure 1.2 illustrates the three layers.

[2] `http://nacl.cr.yp.to`
[3] `http://msgpack.org`
[4] `https://developers.google.com/protocol-buffers/`
[5] such as Ruby's marshalling support: `http://ruby-doc.org/core/Marshal.html`

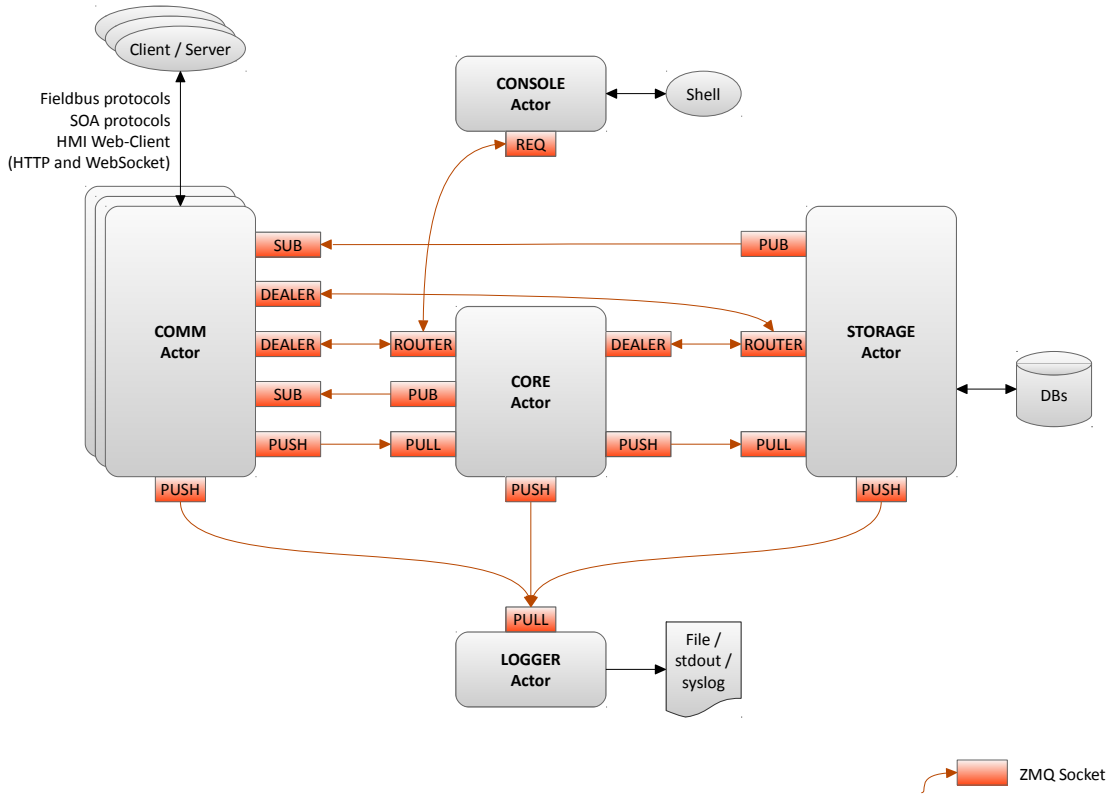


Figure 1.1: Roadster's software architecture

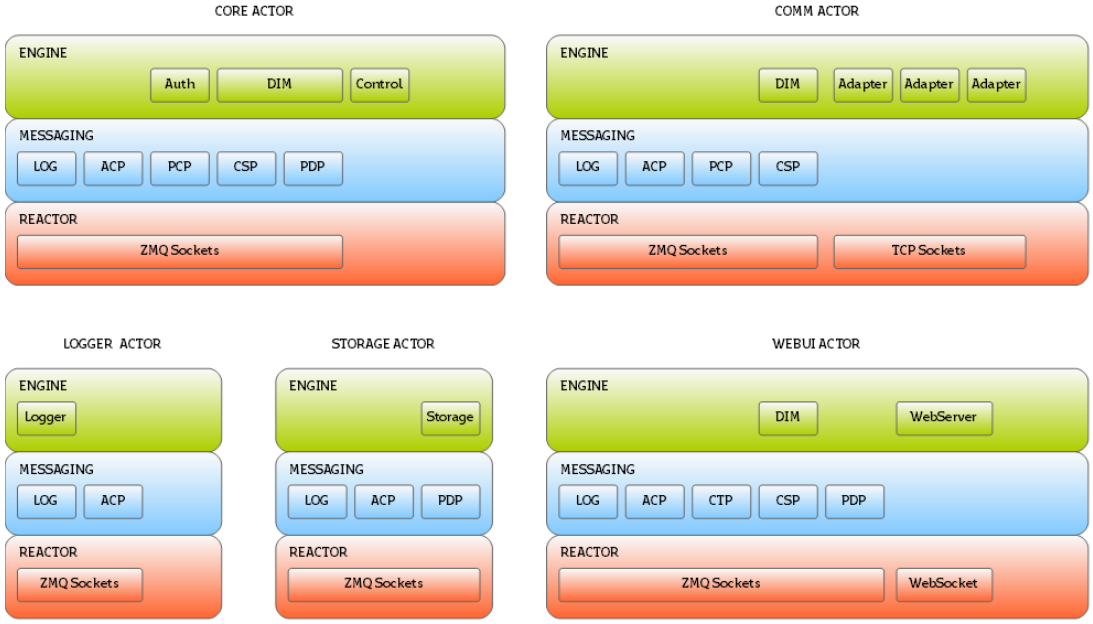


Figure 1.2: Roadster's communication layers

TODO use PDF

Goals

TODO mandatory 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. Highly available OPC UA server

Chapter 2

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.

Priorities

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

1. clustering
2. single-level 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.

Functional

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

Cluster

This requirement is to allow running Roadster on multiple nodes in a hierarchical topology. It could also be called "Multi-node CSP", since extending the Clone State Protocol will be a central part of it. But it's definitely not all that's needed to add cluster functionality.

The DIM must be kept in sync across all nodes. 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

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

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

Persistence Synchronization

This is about the synchronization of the TokyoCabinet databases. Data flow is from south to north (towards the root node), so the root node collects and maintains 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, the delta of the newly added data to the database can be bubbled up. Sending just the delta should be possible since keys in the database contain timestamps.

This is not the same as 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.

Security

TODO common concern about SCADA systems lacking security

- transport needs to be secure (encrypted and authenticated)
- TODO verify requirements with Andy (we didn't really discuss this during the meeting)
- this requirement comes as the last mandatory goal not because it's insignificant, but because it's easy to enable transport level security on ZMQ sockets, and it would just interfere with the previous development

OPC UA HA

- provide standardized interface upwards from HA pair

Use Cases

TODO maybe there are any?

Non-Functional Requirements

Simplicity

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

1. KISS principle – "Keep it simple, stupid." Simplicity is favored, unnecessary complexity should be avoided.
2. No premature optimization since it's the root of all evil.[1]

Testing

- we write unit tests for our own contributions
- we test the integrated result in a close-to-reality setup

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

[1] Quote 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 3

Methodology

TODO what have we done to arrive at the goal (should be reproducible)
TODO this is probably what we know as "Concept"

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

Port to new ØMQ library

TODO justify why port is needed right at the beginning (exclude faults from unmaintained ffi-rzmq 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-rzmq with CZTop

Cluster

TODO describe scribble (chp.pdf)

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 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?

DIM Synchronization

TODO election/design of appropriate protocol

The Existing CSP in a Nutshell

This is a brief introduction/refresher 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[1]. 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

Here are the possible variants:

Variant 1. Sync self-subtree only:

[1] `FIXME`

- 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

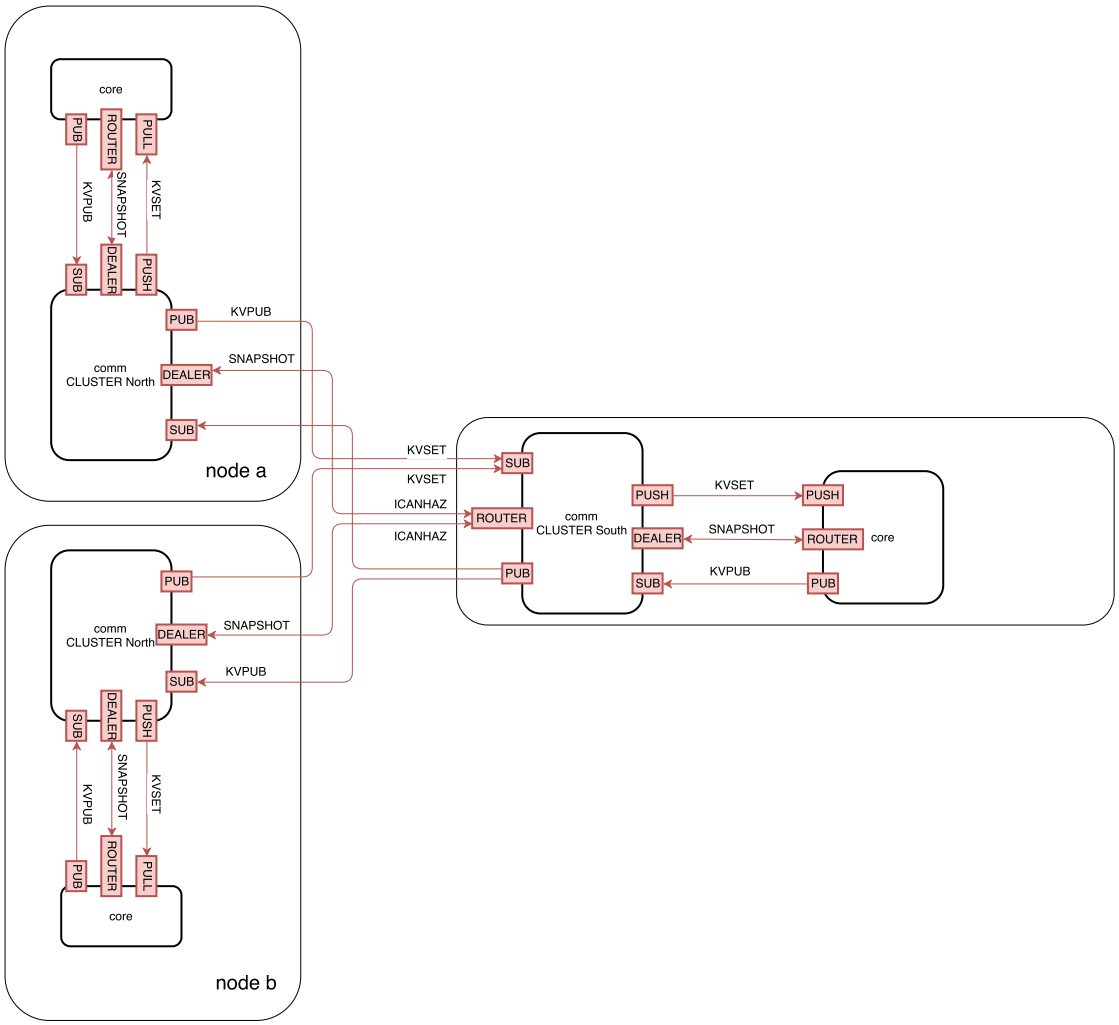
Variant 2 will be the first step. Variant 3 will be the second step, if at all.

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.:



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)

```
# * 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 # Domain::Model
end # Roadster
```

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)

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.

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 alrogrithm 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.

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

Binary Star in a Nutshell

TODO explain binary star
TODO two conditions (no heartbeats, and vote)

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.

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 actve 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 protocol 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.

Side benefit: Rolling Upgrades

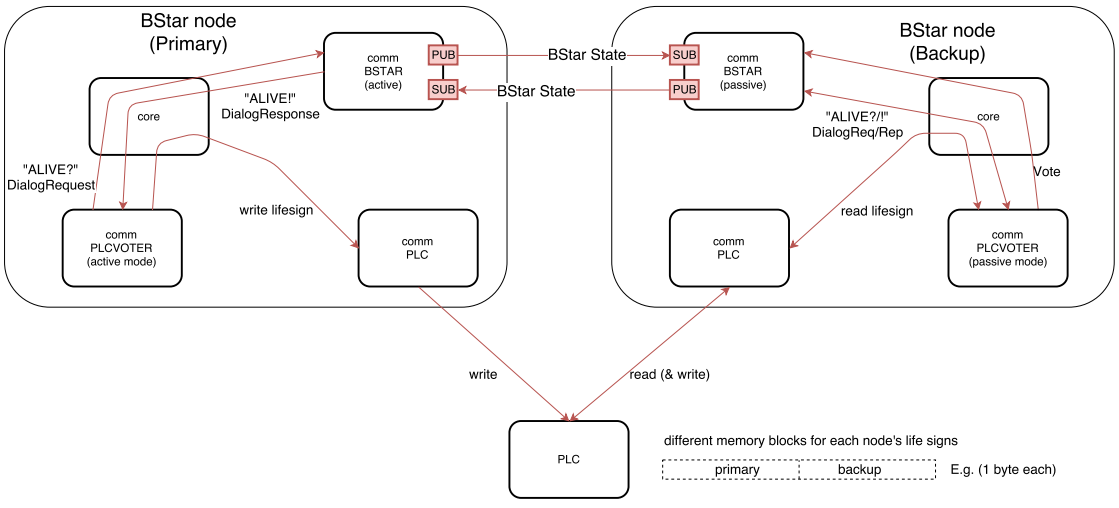
TODO useful for upgrades (maybe in Discussion)

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 protocol 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



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

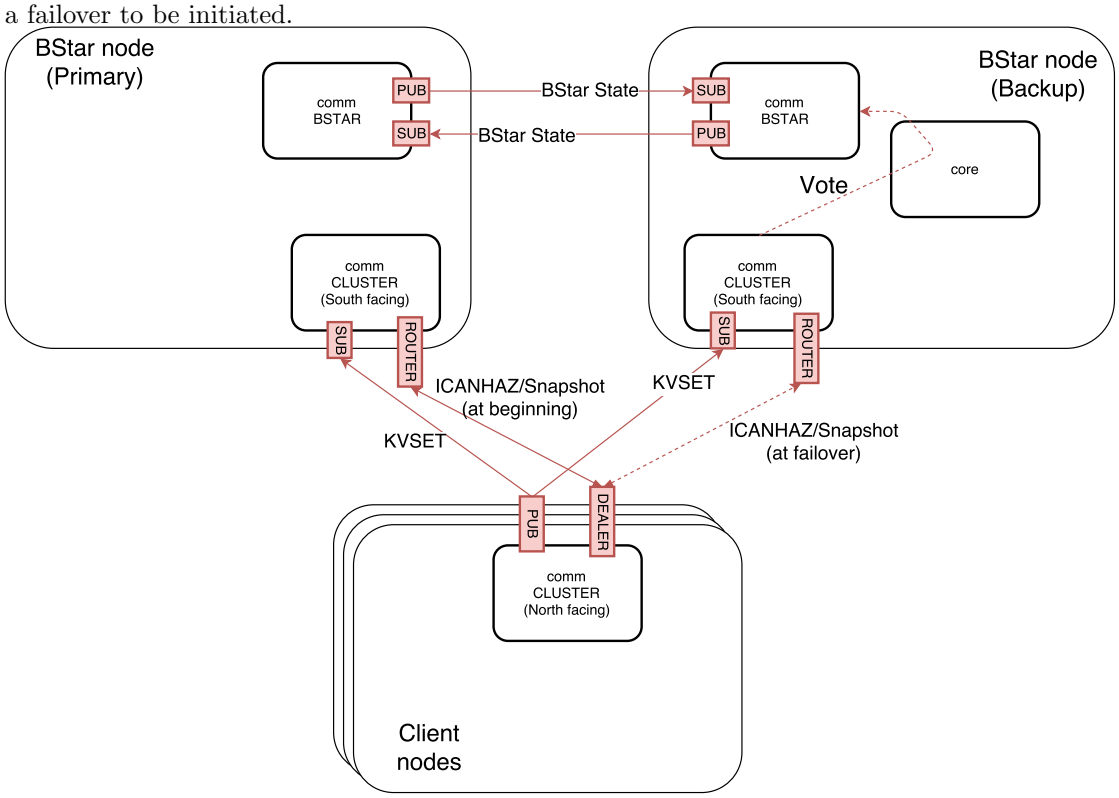
TODO describe why this won't cause a failover and thus can't be handled, as mentioned above

Supporting Different PLCs

TODO: adapter

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.



Persistence Synchronization

The persisted data needs to be bubbled up towards the root node.

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.

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").

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)

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 reliablity 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.

Security

TODO This is the optional goal.
TODO briefly describe ØMQ's security features, what's left for us to decide (key destribution, if at all)
TODO how it can be verified (-¿ using wireshark)
TODO why curvezmq is awesome (confidentiality, integrity, availability (server keeps no state until client is fully authenticated)

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

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

Chapter 4

Results

TODO what are the results (without discussing them)
TODO these is probably the "Implementation"

Port

TODO explain results here

Cluster

TODO explain results here

High Availability

TODO explain results here

Single Level HA

TODO explain results here

Multi Level HA

TODO explain results here

Persistence Synchronization

TODO explain results here

Security

TODO explain results here

OPC UA Interface: High Availability

TODO explain results here

Chapter 5

Discussion

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

Value Added

TODO what's better than before

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.

Business Benefits

TODO potential applications (UeLS powered by Roadster?)

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 6

Conclusion

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

Bibliography

[1] B. FIXME Batsov. *Ruby Style Guide*. URL: https://github.com/bbatsov/ruby-style-guide.

[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.

Part II

Appendix

Appendix A

Self Reflection

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

Appendix B

Task Description

TODO here goes the printed, signed, and scanned Task Description

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

Organization

TODO roles, how we organize ourselves and how we communicate with each other

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

Project Management Software

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