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

Initial Situation

TODO describe initial situation, not too technical

Roadster is a next generation monitoring application.

Software Development Process

TODO describe decision to use RUP/Scrum
TODO maybe describe what project management tools we'll be using

Personal Goals

TODO describe personal goal: the cztop-patterns gem

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

TODO what's this thesis about

Motivation

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

Initial Situation

TODO What's Roadster and its goals

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 [1, 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.

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 Roadster architecture

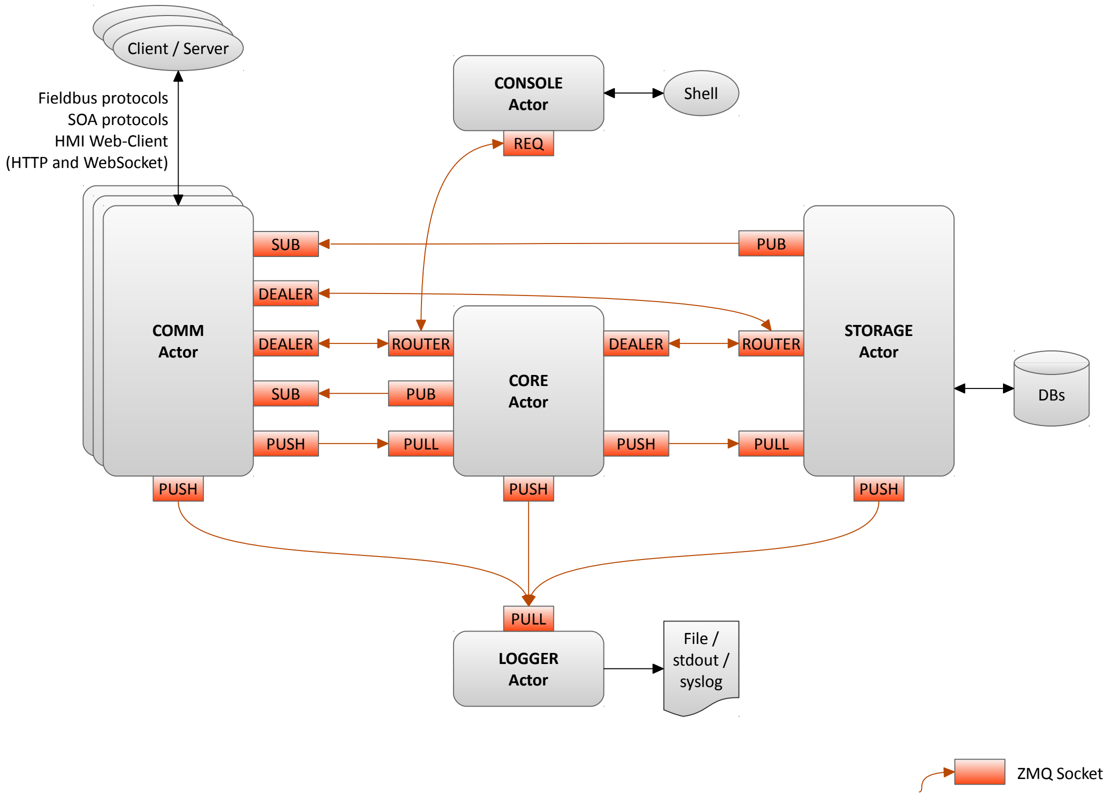
[1] `http://zguide.zeromq.org/`

[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`



Goals

TODO mandatory goals

Optional Goals

TODO optional goals

Chapter 2

Requirements

TODO the requirements

Priorities

In descending priority:

1. multi-node CSP
2. single-level HA
3. multi-level HA
4. persistence synchronization
5. security
6. OPC UA HA (optional)

The following sections explain the requirements in greater detail.

Functional

Cluster

This could also be called ”Multi-node CSP”.

- this is to allow running Roadster in a hierarchical setup
- new COMM actors for inter node communication
- usually 2 (or 3) levels of Roadster nodes
- common cases:
  - - single level, single node (legacy)
  - - single level HA
  - - multi level, HA at root node only
- exotic cases:
  - - multi level, HA at bottom
  - - multi level, HA in middle

- every subtree can live on autonomously
- only node A has write access to values on A (to avoid uncertain situations involving race conditions), e.g.:
  - - a forced value coming from the web UI comes through a command,
  - - routed to the relevant node, where it is applied,
  - - and then synced (up via DEALER and down via PUB, we suppose)
- KISS

```
     C
   /   \
  A     B
```

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.

TODO the kinds of failures we want to be able to handle: exactly hardware/software failure of the primary node, and network failure (stated by the Task Description)

Multi Level HA

This is where a node pair is the parent of one or more other nodes (subnodes).

TODO the kinds of failures we want to be able to handle: exactly hardware/software failure of the primary node, and network failure (stated by the Task Description)

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.

- autonomous
- not same as CHP
- 100% consistency is not important
- data only flows from bottom to top
- TC keys contain timestamp

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

TODO the NFRs

Testing

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

Coding Guidelines

- basically Ruby style guide[1]
- 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 (just like [5])
- align multiple assignments so there's a column of equal signs

[1] https://github.com/bbatsov/ruby-style-guide

[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"

Port to new ZMQ 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 FD and EVENTS in CZTop
TODO explain concept of exchanging ffi-rzmq with CZTop

Cluster

TODO describe scribble (chp.pdf)

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
TODO explain Clustered Hashmap Protocol (?)

- PCP: use DIM to know node tree and determine next hop for (dialog or fire+forget) messages
- decide on sync variant
  - variant 1
    - always sync on self-subtree only
    - con: no copy of remaining tree
  - variant 2:
    - always sync on complete tree
    - get snapshot and merge own subtree
    - (this should probably be the first step)
  - variant 3:
    - make it configurable: either sync on subtree or complete tree
    - (this should probably be the second step, if at all)
- we need two new COMM actors
- they sync between super node and sub 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

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

TODO we have two different kinds of HA
TODO explain how the failures we're required to be able to handle can be handled
TODO expalin similarities between the two kinds of HA

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)
- life sign from one node to the other through some continually updated PLC value
- mark active HA peer in DIM

TODO integrate findings from scribble (SL-HA.pdf)

Multi Level

TODO explain why is this one different from SL-HA
TODO Finding consensus should be easier here, as it's closely related to the CHP described in the zguide.
TODO integrate findings from scribble (ML-HA.pdf)

Persistence Synchronization

- super node requests for delta of TC periodically

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. ZMQ'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, is similar to what's used for the 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, to quote Donald Knuth, "Premature optimization is the root of all evil". If this turns out to be an issue in a productive deployment, like over a mobile link, a future version can switch to multicast. ZMQ 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 ZMQ-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 briefly describe ZMQ's security features, what's left for us to decide (key destribution)
TODO how it can be verified (-¿ using wireshark)

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, we're the best and everything is awesome

Bibliography

[1] A. Dwerak, 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

ZMQ

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