ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

HOW TO DESIGN & IMPLEMENT A MODERN COMMUNICATION MIDDLEWARE BASED ON ZeroMQ

J. Lauener*, W. Sliwinski†, CERN, Geneva

Abstract

In 2011, CERN's Controls Middleware (CMW) team started a new project aiming to design and implement a new generation equipment access framework using modern, open-source products. After reviewing several communication libraries [1], ZeroMQ [2] was chosen as the transport layer for the new communication framework. The main design principles were: scalability, flexibility, easy to use and maintain. Several core ZeroMQ patterns were employed in order to provide reliable, asynchronous communication and dispatching of messages. The new product was implemented in Java and C++ for client and server side. It is the core middleware framework to control all CERN accelerators and the future GSI FAIR [3] complex. This paper presents the overall framework architecture; choices and lessons learnt while designing a scalable solution; challenges faced when designing a common API for two languages (Java and C++) and operational experience from using the new solution at CERN for 3 years. The lessons learnt and observations made can be applied to any modern software library responsible for fast, reliable, scalable communication and processing of many concurrent requests.

INTRODUCTION

A control system needs a performant communication infrastructure offering a reliable exchange of data between distributed processes. Each process acts either as a client or as a server, or even both. The communication capability is provided by a middleware software framework, composed of client & server parts, exposing a public API to the application layer.

Technology Evolution

For the needs of the CERN accelerator control system, a middleware framework called RDA (Remote Device Access) [4] was designed and its first version was implemented in 2000. Initially, RDA was built on top of the CORBA transport layer. However, after using the CORBA based solution for more than 10 years, a number of outstanding issues were identified: poor scalability and heavy use of system resources (CPU & memory). In 2011, it was decided to perform a market survey [1], aiming to find a modern transport library, replacing completely CORBA and providing the required scalability and performance levels. The review process selected ZeroMQ as a transport library for the new, major version of RDA, called subsequently RDA3.

Requirements for RDA3

A set of functional & technical requirements was formulated for RDA3. All major functional requirements remained the same as for the previous RDA2. However, based on the operational experience, certain technical aspects (e.g. asynchronous transport) became obligatory for the new RDA3.

Here are the most important functional requirements:

- Support for required data types: scalars, strings, data structures, multi-dimensional arrays of scalars/strings/data structures
- Access to remote resources based on the device-property model
- Provide sync & async *Get* call (read data)
- Provide sync & async *Set* call (write data)
- Provide *Subscribe* call (monitor data changes)
- Guaranteed, ordered execution of requests on the server-side and ordered reception of results
- Consistent implementation for C++ and Java

Additionally, equally important technical requirements:

- Fully asynchronous communication
- Good scalability, exceeding by far RDA2
- Quality of Service (QoS): timeout management, message queues, thread management policies
- Low usage of memory and system resources
- Portable solution, with minimal external dependencies, which can be easily adopted to any platform
- Intuitive, extendable, safe and easy to use public API.

RDA3 Overview

In the CERN context, RDA3 as the middleware framework, provides transparent access to equipment, following the device-property model used at CERN (i.e. the means to access remote resources) [4]. In this model access points are represented as device-property pairs. A device is an abstraction of the underlying equipment. A property represents an operation that can be performed on the device.

The framework supports two communication paradigms:

- *request/reply*: client can either read from (*Get* call) or write to (*Set* call) an access point.

* Joel.Lauener@cern.ch
† Wojciech.Sliwinski@cern.ch

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

- *publish/subscribe*: client can subscribe (*Subscribe* call) to an access point and get a notification from the server whenever the value changes.

The framework provides client & server parts and it is available in Java and C++. The integration with needed infrastructure services such as the Directory/Naming service or the Authorization/Authentication service is done through dependency injection, allowing it to be easily adaptable for use outside of CERN [3].

ZeroMQ

ZeroMQ is an open-source networking library originally developed by iMatix under the LGPLv3 license. Following the vision of its main author Pieter Hintjens the evolution and maintenance of ZeroMQ is driven by an active community.

The library provides a compact and simple socket API similar to BSD sockets. ZeroMQ sockets can be used to establish in-process, inter-process or inter-host (using TCP or multicast) communication. It supports various types of paradigms from simple REQ/REP to PUB/SUB, task distribution and fan-out.

On the wire, it uses the ZMTP protocol (*ZeroMQ Message Transfer Protocol*) [5]. Smart use of message batching, asynchronous communication and support for zero copy (hence the 'Zero' part in its name) makes it one of the most efficient libraries for creating distributed applications.

ZeroMQ core is written in C/C++ but it has bindings and/or native ports for most modern languages and operating systems [6].

Test Driven Development (TDD)

Development of RDA3 was largely test driven. From day one testable code was written. Not only did TDD result in robust and maintainable code, but in overall it also contributed to a better internal design through the use of injectable interfaces.

Initially TDD incurred an extra cost for the development. Writing and maintaining good tests with same code quality as production code takes time. After 3 years of operation the gap was largely compensated thanks to reduced support and ease of maintainability.

ARCHITECTURE

The framework is split into two major parts (see Fig 1):

- *The Transport Layer*: abstracts and hides the underlying networking library ZeroMQ.
- *The Business Layer*: implements the device-property model, connection management, task scheduling, priority management and error recovery.

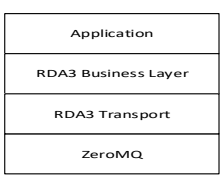

Figure 1: Overall RDA3 architecture.

Transport Layer

The Transport Layer allows bi-directional asynchronous communication between the peers (client or server). It also manages connections using the heartbeat mechanism.

Dispatcher Thread: At its core the Transport Layer handles all communication through a single thread called the Dispatcher Thread (see Fig 2).

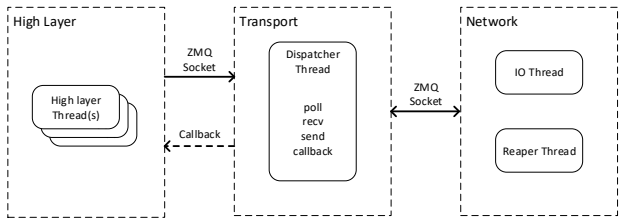

Figure 2: Dispatcher Thread.

This thread has 4 responsibilities:

- Poll for incoming messages from the Business Layer (requests) or from the ZeroMQ network threads (replies);
- Send requests to the ZeroMQ network threads;
- Call back the Business Layer when a reply is received;
- Manage connection by checking and sending heartbeat messages.

The use of a single thread enables a lock-free design through the use of socket polling. Its main task is deceptively simple: it looks at the message destination and dispatches it to the correct socket. It is critical that the dispatcher thread runs without blocking or slow I/O operations.

Connection management: The Transport Layer performs bi-directional asynchronous heartbeats between the peers. The implementation is symmetrical between the client and the server. Any message counts as a heartbeat message. If a remote peer doesn't send any messages during a given duration (1 second by default), it sends an empty heartbeat message. If a peer doesn't receive any messages during a given duration (10 seconds by default) it considers itself disconnected from the remote one.

The Transport Layer doesn't perform reconnection to a peer. Once it detects a disconnection it cleans up all associated resources and notifies the Business Layer, which handles all the reconnection logic.

Network request/reply: ZeroMQ has a few useful socket combinations for achieving request/reply communication:

- REQ/REP: This is a pure request/reply; for each request the remote peer must send a reply. This pattern doesn't support asynchronous communication and is rarely used in scalable applications.
- DEALER/DEALER: The DEALER socket allows for asynchronous request/reply. Each peer can send a

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

number of request/reply messages it wants. This pattern is mostly used for 1-1 communication.

- DEALER/ROUTER [7]: The ROUTER socket works like a DEALER, but in addition it fans out to many peers. When a message is sent/received on a ROUTER socket the first frame contains the client ID and is used to route the message to the correct peer. This is the pattern of choice for server to n-client communication.

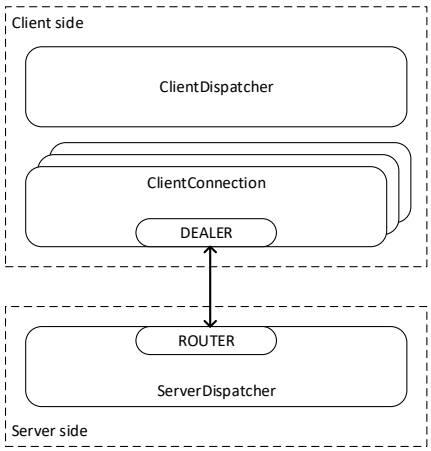

Figure 3: Client-server ZeroMQ sockets.

RDA3 uses the DEALER/ROUTER pattern. As seen on Fig 3 the server side creates only one single ROUTER socket to handle all client communication. On the client side one DEALER socket is created per server connection. When a client detects a disconnection it simply closes the corresponding DEALER socket.

Network publish/subscribe: Initially it was planned to use the ZeroMQ PUB/SUB pattern to implement the subscription facility of the device-property model. In this design one channel (DEALER/ROUTER) was used for request/reply and a second channel (PUB/SUB) was used for subscriptions. This is a perfectly correct approach however it was decided to drop the PUB/SUB channel and to use the same asynchronous request/reply channel for all operations. There were several reasons for this decision:

- The device-property model has the concept of the first update. This means when a client subscribes to an access point, the server must always send an initial value to the client. This cannot be achieved using PUB/SUB as it is impossible to address a specific peer.
- When a client tries to establish a subscription, the server can reject it. This mechanism is used by the Authorization service. PUB/SUB doesn't provide any means to reject a given peer.
- There is no scalability/performance gain in using PUB/SUB sockets unless multicast [8] is used. Multicast is mostly used for video broadcast and is not applicable to control an equipment.

To keep the design simple and lightweight we decided not to use PUB/SUB and handle all communication on the same DEALER/ROUTER channel.

Communication with the Business Layer: In order to send a message to the Transport Layer the ZeroMQ's PUSH/PULL pattern is used. This pattern allows blocking unidirectional communication with a high-water mark. In case of flooding the Business Layer would get blocked until the Dispatcher Thread can process an another message. This allows to put backpressure on the Business Layer in case of overload.

Sending back messages to the Business Layer is done through a programmatic callback. It must be noted that this callback is executed inside the Dispatcher Thread so it is critical to have a fast and predictive execution.

Business Layer

The Business Layer implements all high-level logic specific to the equipment access and it integrates with the Transport Layer, Directory service and Authorization service. It heavily relies on dependency injection, so all services are optional and customizable.

The Business Layer doesn't make use of ZeroMQ for several reasons:

- It must be agnostic of the underlying communication library;
- Tasks (messages in the Business Layer) need to have references to pooled objects such as access point or source address. ZeroMQ messages can only contain byte buffers;
- Simple in-process native queues are faster than ZeroMQ in-process sockets.

Group Task Scheduler: Task management is performed by the so-called Group Task Scheduler, an in-house generic task scheduler that supports overflow based on an arbitrary grouping criterion (see Fig 4).

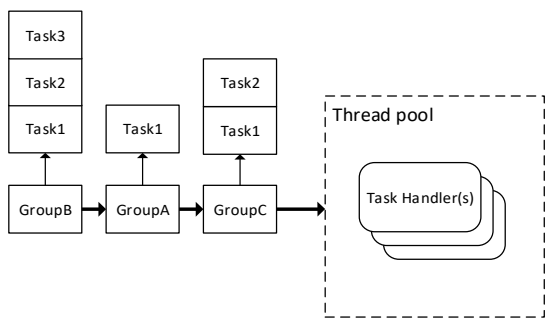

Figure 4: Group Task Scheduler.

The criterion for grouping is a string, which allows for a lot of flexibility. For example, the server groups tasks per device, where the client groups tasks per server ID or subscription ID. All tasks for a given group are executed in a FIFO (*First-In First-Out*) order, which insures task order is respected for a given group.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

Serialization: At the time when RDA3 was implemented, ZeroMQ didn't provide any serialization scheme, leaving this up to the users to choose one. Therefore, several third-party solutions were evaluated for message serialization:

- Google Protocol Buffers [9]: interface-based (proto3); uses code generation.
- CORBA serialization [10]: interface-based (IDL); uses code generation, part of the CORBA stack.
- MessagePack [11]: dynamic (no IDL); comes with many dependencies; slow for double arrays.
- Apache Avro [12]: dynamic (no IDL); has a separate schema; comes with many dependencies.

RDA3 exposes narrow public API, with small number of operations, therefore an interface-based serialization requiring an additional code generation seems to be an unnecessary complication. Also, most serialization libraries come with additional dependencies, which could be potentially problematic to port to a new platform. For those reasons, a custom library for data serialization was developed in-house. It supports binary, string and CSV serialization. It's a simple, lightweight and fast library with no dependencies. The library is composed of serializers and a generic data container used by the application tier.

In order to evaluate the performance of the custom CMW serializer a benchmark was setup using a data object with arrays (int, double, string) of 40 elements each. Next, an average serialization/deserialization execution time was computed (see Fig 5).

Serializer	Time (ms)	Size (bytes)
CMW	422	1071
MsgPack	1342	932
CORBA	1108	1152

Figure 5: Serialization benchmark (100'000 cycles).

Header and body: When a message is received from the Transport Layer it needs to be deserialized in order to add it to the correct group in the task scheduler. As mentioned in the Transport Layer subsection, this is done inside the Dispatcher Thread callback and it must take a predictable amount of time in order to let the Dispatcher run smoothly. As the user payload of a message cannot be determined in advance, the content of the message is split into two parts:

- The Header: contains the type of task and address information (either a remote host or a device/property);
- The Body: contains the payload of the message.

As soon as a message is received its header is deserialized in order to determine its group and the whole message is sent to the task scheduler. The body is lazily deserialized only once the payload is needed. This approach ensures the code executed in the Dispatcher Thread runs at a constant time. Moreover, lazy deserialization of the body helps to reduce the negative impact on the CPU in case of an overflow. Indeed, in case of an overflow the task scheduler drops messages, so the body doesn't need to be deserialized at all.

DESIGNING FOR JAVA AND C++

Java and C++ are both object-oriented programming languages reminiscent of the C family. During the development of RDA3 an effort was made to keep a simple and symmetrical API and implementation whenever possible (see Fig 6). This helped to reduce the design effort and to reduce the cost of support as fixing a bug on two similar code bases is much faster.

Java

```
ClientService client =
  Rda3Factory.createClientService();
AccessPoint accessPoint =
  client.getAccessPoint("dev", "prop");
AcquiredData acqData = accessPoint.get();
Data data = acqData.getData();
System.out.println(data);
```

C++

```
std::auto_ptr<ClientService> client =
  Rda3Factory::createClientService();
AccessPoint& accessPoint =
  client->getAccessPoint("dev", "prop");
std::auto_ptr<AcquiredData> acqData =
  accessPoint.get();
const Data& data = acqData->getData();
std::cout<<data.toString()<<std::endl;
```

Figure 6: Example use of Java and C++ client API.

Each language has its perks which inevitably influence the symmetrical implementation. The main caveat is the richness of the C++ vocabulary to define ownership and immutability against the lack of support in Java.

Memory Management

Java has a garbage collector that handles all the allocations/deallocations, where C++ gives full control of the memory management through rich pointer semantics. RDA3 C++ API never exposes raw pointers, instead it heavily relies on 'std::auto_ptr' and 'std::shared_ptr'.

Immutability

C++ has the 'const' keyword. Java has no equivalent besides the weaker 'final' keyword. Java's 'final' can only make a reference immutable, where in C++ 'const' can be used to express immutable reference and object. Immutability is critical for writing safe, concurrent applications.

In order to keep the API simple and symmetrical it was decided to not use wrappers and builders to achieve immutability in Java. The information found in the API documentation is used as a contract with the end-user to define when an object can be muted or not. Most of the time after an object is passed to the library the user is not allowed to modify it afterwards.

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

Unsigned Types

Java has no unsigned types. To keep interoperability between Java and C++ the data container doesn't support unsigned types.

Utilities

Java is known for its full-fledged SDK. The C++ implementation uses the Boost [13] library. Boost is never exposed to the end-user. Common utilities are wrapped inside a small library that is also available to the end user for basic operations such as string manipulation and time measurement.

RUNTIME DIAGNOSTICS

In order to administer remotely RDA3 servers, an administrative interface is provided. Through this interface it is possible to collect useful runtime information such as counters, subscription information and client information. The interface can also be used to send commands to the server. The framework comes with a default set of commands and a plugin-in architecture makes it easy to extend the admin interface capabilities. For example, at CERN, commands to perform operations on the authorization extension (*RBAC* [14]) were added. Access to the admin interface can be done either via a programmatic API or using a dedicated graphical user interface called *CMW-Admin*.

In the initial design, it was planned to create a separate pair of DEALER/ROUTER sockets to carry all administrative messages. But considering that the Grouped Task Scheduler can be used to separate administrative messages from normal ones it was decided to use a single channel for all types of messages.

MIGRATING FROM CORBA TO ZeroMQ

The previous major RDA version, namely RDA2, which is based on CORBA is still operational within the CERN infrastructure.

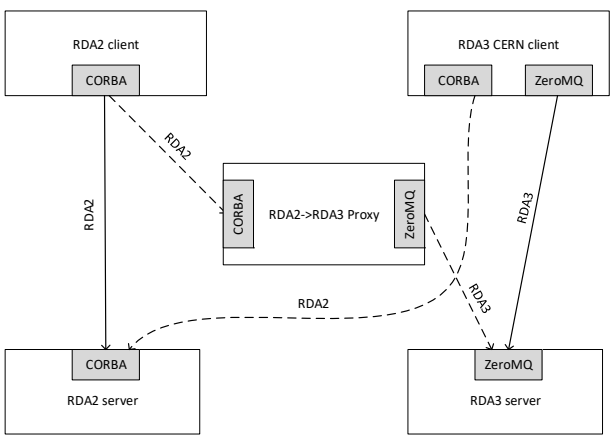

Figure 7: Interoperability between RDA2 and RDA3.

During RDA3 deployment phase it was critical to find a solution to let the two non-compatible middlewares coexist during the migration period, which would span over several years (see Fig 7).

Client Side

On the client side, a CERN-specific RDA3 client was developed that can communicate with both middlewares. To achieve this the RDA2 client was wrapped behind the RDA3 interface. The smart client then, depending on the type of the server, chooses the correct implementation.

Server Side

On the server side, it was decided not to include the RDA2 server together with the RDA3 server. The reason behind this decision is that in our infrastructure it is quite difficult to upgrade massively servers. To still allow old RDA2 clients to communicate with new RDA3 servers, a RDA2-to-RDA3 gateway called *Proxy* [15] was introduced. Each Proxy is manually configured to handle a given set of RDA3 servers. Configuration is not automatic in order to prevent proliferation of old RDA2 clients.

CONCLUSION

After using RDA3 in operation for more than 3 years, to control all CERN accelerators, we can definitely confirm that choosing ZeroMQ as a networking library was the right decision. On many occasions, it was clearly visible that RDA3 based on ZeroMQ scales much better and can smoothly handle high data loads and even bursts of requests, which was not the case for RDA2 based on CORBA. This is possible thanks to fully asynchronous ZeroMQ transport and event-driven architecture of RDA3.

In the future, it is planned to investigate several ZeroMQ features: socket monitor for quick connection clean-up; heart-beat mechanism for connection management; multicast messaging.

RDA3 was designed to use only a few dependencies and to comply with modern API design principles. It is also intention of the CERN CMW team to open-source the RDA3 framework in the future.

For questions please email to: cmw-info@cern.ch

ACKNOWLEDGEMENT

Over a period of last 5 years, several people significantly contributed to the design and implementation of RDA3. Their hard work and commitment made possible the success of RDA3. We would like to acknowledge their contributions (in alphabetical order):

- **Felix Medina Chapilliquen**
- **Andrzej Dworak**
- **Radoslaw Orecki**
- **Vitaliy Rapp (GSI)**
- **Ilia Yastrebov**
- **Wojciech Zadlo**

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

APPENDIX: ONE PAGE RDA3 CLIENT-SERVER APPLICATION

A C++ server that supports get/set and subscribe.

```
#include <iostream>

#include <cmw-rda3/server/service/ServerBuilder.h>
#include <cmw-data/DataFactory.h>

#include <cmw-util/TimeUtils.h>
#include <cmw-util/StringUtils.h>

using namespace cmw::util;
using namespace cmw::data;
using namespace cmw::rda3::server;
using namespace cmw::rda3::common;

class : public RequestReplyCallback
{
public:
    void get(std::auto_ptr<GetRequest> request)
    {
        std::auto_ptr<Data> data=DataFactory::createData();
        data->append("hello", "client");
        request->requestCompleted(AcquiredData(data));
    }

    void set(std::auto_ptr<SetRequest> request)
    {
        std::cout<<"set:"<<request->getData().toString();
        std::cout<<std::endl;
        request->requestCompleted();
    }
} rrCallback;

class : public SubscriptionCallback
{
public:
    void subscribe(SubscriptionRequest& request)
    {
        SubscriptionCreator& creator=request.accept();
        creator.startPublishing();
    }

    void unsubscribe(const Request& request)
    {
    }

    void subscriptionSourceAdded(
      const SubscriptionSourceSharedPtr& subscription)
    {
    }

    void subscriptionSourceRemoved(
      const SubscriptionSourceSharedPtr& subscription)
    {
    }
} subCallback;

int main(int argc, const char* argv[])
{
    std::auto_ptr<ServerBuilder> builder=
      ServerBuilder::newInstance();

    builder->setServerName("Rda3DemoServer");
    builder->setRequestReplyCallback(rrCallback);
    builder->setSubscriptionCallback(subCallback);
    std::auto_ptr<Server> server=builder->build();

    server->start(false);

    while (true)
    {
        std::list<SubscriptionSourceSharedPtr> subs=
          server->getSubscriptionLookup().getSubscriptions();

        std::list<SubscriptionSourceSharedPtr>::iterator it;
        for (it=subs.begin(); it != subs.end(); ++it)
        {
            std::auto_ptr<Data> data=DataFactory::createData();
            data->append("hello", "client");

            SubscriptionSource & sub = **it;
            sub.notify(AcquiredData(data));
        }

        TimeUtils::sleep(TimeUtils::Time(1, TimeUtils::sec));
    }
}
```

A Java client that performs get/set and subscribe.

```
import cern.cmw.data.Data;
import cern.cmw.data.DataFactory;
import cern.cmw.rda3.client.core.AccessPoint;
import cern.cmw.rda3.client.service.ClientService;
import cern.cmw.rda3.client.subscription.Notification;
import cern.cmw.rda3.client.subscription.SubscriptionQueue;
import cern.cmw.rda3.common.Rda3Factory;
import cern.cmw.rda3.common.data.AcquiredData;

public class ClientDemo {
    public static void main(String[] args) throws Exception {
        ClientService client=Rda3Factory.createClientService();

        AccessPoint accessPoint=
          client.getAccessPoint("device", "property");

        AcquiredData getResult = accessPoint.get();
        System.out.println("Get: "+getResult);

        Data data = DataFactory.createData();
        data.append("hello", "server");
        accessPoint.set(data);

        SubscriptionQueue subscription=accessPoint.subscribe();
        while (true) {
            Notification notification=subscription.poll();
            System.out.println("Notification: "+notification);
        }
    }
}
```

ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-MOBPL05

REFERENCES

[1] A. Dworak *et al.*, "Middleware trends and market leaders 2011", ICALEPCS 2011, Grenoble, France, 2011.

[2] ZeroMQ, http://zeromq.org

[3] V. Rapp *et al.*, "Controls Middleware for FAIR", PCaPAC 2014.

[4] N. Trofimov *et al.*, "Remote Device Access in the new CERN accelerator controls middleware", 2001, ICALEPCS 2001, San Jose, USA, 2001.

[5] ZMTP, https://github.com/zeromq/zmtp

[6] ZeroMQ language bindings, http://zeromq.org/bindings:_start

[7] ZeroMQ DEALER/ROUTER pattern, http://zeromq.org/tutorials:dealer-and-router

[8] IP-Multicast, https://en.wikipedia.org/wiki/Multicast

[9] Google Protocol Buffers, https://developers.google.com/protocol-buffers/

[10] CORBA, http://www.corba.org/

[11] MessagePack, http://msgpack.org/index.html

[12] Apache Avro, https://avro.apache.org/

[13] Boost C++ libraries, http://www.boost.org/

[14] S. Gysin *et al.*, "Role-Based Access Control for The accelerator control system at CERN", ICALEPCS 2007, Knoxville, USA, 2007.

[15] W. Sliwinski *et al.*, "Middleware Proxy: a Request-Driven Messaging Broker for High Volume Data Distribution", ICALEPCS 2013, San Francisco, USA, 2013.