ISBN: 978-3-95450-193-9 doi:10.18429/JACoW-ICALEPCS2017-THPHA038

UPGRADE OF THE CERN RADE FRAMEWORK ARCHITECTURE USING RabbitMQ AND MQTT

O. O. Andreassen, F. Marazita, M. K. Miskowiec, CERN, Geneva, Switzerland

Abstract

AMQP (Advanced Message Queuing Protocol) was originally developed for the finance community as an open way to communicate the vastly increasing over-the-counter trace, risk and clearing market data, without the need for a proprietary protocol and expensive license. In this paper, we explore the possibility to use AMQP with MQTT (Message Queue Telemetry Transport) extensions in a cross platform, cross language environment, where the communication bus becomes an extendible framework in which simple/thin software clients can leverage the many expert libraries at CERN.

INTRODUCTION

The Rapid Application Development Environment (RADE) was initially developed to make it possible to interface LabVIEW™ based equipment and software with the CERN technical infrastructure. As part of this implementation, a multi-tier communication layer was introduced called RADE Services [1].

The current implementation of the RADE Services are based on custom-coded Java application interfaces linking the RADE client interfaces with an Apache Tomcat Web Server [1]. Despite the stability and performance of this implementation, there are several issues that need to be addressed in order to improve the scalability, reusability and cluster performance of the service.

Custom code forces developers to rewrite and sometimes re-design whenever the dependent libraries change. Maintenance has proven to be time-consuming and with heavy traffic, the solution doesn't scale well.

To address this, we started looking at industrial solutions working in a more efficient and compartmentalized way, reducing code redundancy. Moreover, in search of better approach it was necessary to uphold two major requirements– platform independence and plugin support for JAVA – LabVIEW™ communication [1].

In this paper, we will show how we try to address this problem by linking most of the various services and interfaces via a commercially supported, well documented software layer called RabbitMQ, and leverage the interoperability by reducing the software complexity and maintenance efforts [2].

CERN Infrastructure

The technical infrastructure at CERN consists of several different front-end devices, databases, sensors and experimental equipment (See figure 1).

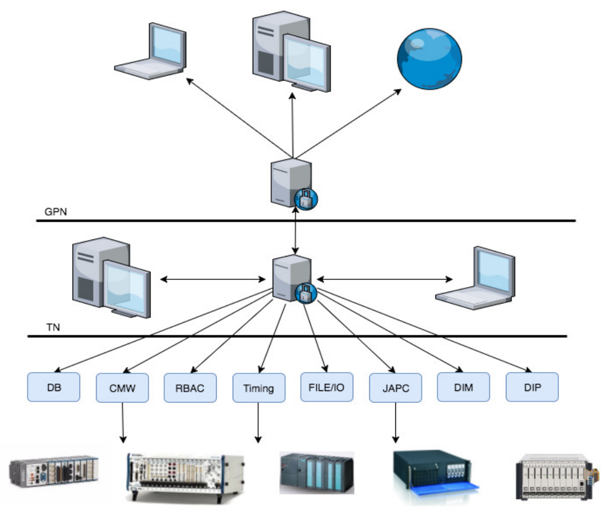

Figure 1 : Simplified view of the CERN Technical Infrastructure

To keep track of the equipment, the CERN Controls Configuration Database (CCDB) holds the information describing the different data interfaces and the relations between the hardware and software. You will also find information describing the network interfaces and every device connected to the CERN technical and general purpose network [2].

Access to the CCDB is provided by the middleware and communication layers.

The Controls Middleware (CMW) framework enables client applications to connect and retrieve data from CERN accelerator equipment [3] [4].

Apart from databases there are also several file systems designed for data storage [5]:

- DFS (Distributed File System)
- NFS (Network File System)
- AFS (Andrew File System)
- EOS

RADE Services

The RADE framework aims to give users a total package for development, maintenance and support through well-defined templates, guidelines and documentation. RADE libraries make use of a distributed architecture (See figure 2), with several application servers hosting dedicated communication and analysis libraries [1].

As an example, the Java API for Parameter Control (JAPC) is a communication layer to control accelerator devices from Java. In the RADE Services stack, JAPC is used as a unified software API where you can get access to most parameters of the CERN accelerators control parameters.

It combines several databases, file formats and inter process communication mechanisms that we make use of in the RADE Services RabbitMQ bridge [1][2].

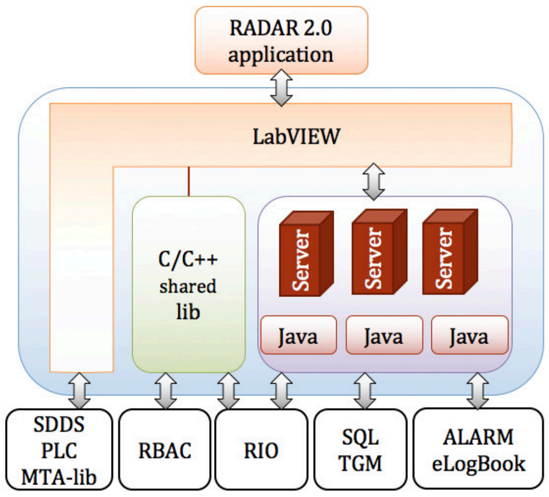

Figure 2: The RADE Framework Distributed Architecture

In addition to JAPC, many different interfaces have been deployed in the RADE Services infrastructure [1]:

- Java API for Parameter Control (JAPC) is an API for control system parameters [1].
- The Injector Control Architecture (InCA) is a service responsible for get, set and subscribe operations through InCA libraries [6].
- Distributed Information Management (DIM) Service provides subscription using the DIM library [7].
- CERN Accelerator Logging Service (CALS) allows LabVIEW™ users access to CERN databases [2].
- PLSLineListener is a project for PLSLine subscription [1].
- LHC Software Architecture (LSA) is responsible for setting new parameters into the LSA database and retrieving trim history [1].
- JAPCPublisher adds functionality to JAPC for publishing data to the CALS database [1][3].
- ServletChecker keeps the availability of the previous projects in the RADE Servers.

Apache Tomcat

Apache Tomcat is an open source web server, developed by the Apache Software Foundation. Apache Tomcat implements several Java EE specifications and provides Java HTTP web server environment where Java web applications can run [8].

RabbitMQ

RabbitMQ is an open source, lightweight message broker with a built-in end to end queueing mechanism that enables applications to share data using a common well defined protocol. It brings benefits such as load balancing and job distribution [2].

RabbitMQ ships with several interfaces that makes it possible to simultaneously cross communicate between multiple servers and clients across different programming languages. This, along with a comprehensive list of tutorials, strong community support and explicit examples, simplifies the overhead of developing dedicated applications since most of the business logic is outsourced and re-used between the applications via service managers [2].

The RabbitMQ broker is built upon the Advanced Message Queueing Protocol (AMQP). It has several extensions such as the Message Queue Telemetry Transport (MQTT) mechanism that provides a powerful and flexible solution for communication between software clients and servers at CERN.

AMQP is an openly published wire specification for asynchronous messaging and standard protocol for message-oriented middleware [9].

MQTT is a machine-to-machine connectivity protocol, designed as a lightweight messaging transport, typically used in the industry for endpoint communication [10].

The RabbitMQ broker provides several excellent management-plugins that works as a HTTP-based APIs where one for example can monitor and manage RabbitMQ clusters through browser-based user interfaces [2].

Apache Kafka

Apache Kafka is an open-source stream processing platform developed by the Apache Software Foundation written in Scala and Java. Kafka is used for building real-time data pipelines and streaming apps. It is scalable, fault-tolerant, fast, and used in production in many companies. The project aims to provide a unified, high-throughput, low-latency platform for handling real-time data feeds [11].

STUDY

Before introducing the new architecture, we studied the existing implementation to explore its advantages and disadvantages. Moreover, we discussed benefits of using broker-based approach over peer to peer communication or usage of Kafka. Our study shows superior results of RabbitMQ over other possible solutions.

Apache Tomcat

The first version of the Java based distributed RADE Services was built upon applications interfacing with an Apache Tomcat Web Server. The LabVIEW™ application interfaces were communicating with the Tomcat server using HTTP POST methods and TCP/IP listeners (See figure 3). This implementation provided a stable and relatively fast interface, but it lacked the scalability and flexibility of today's modern message brokers. In addition, the implementation had a lot of code repetition and redundancy. Several separate static wrappers were used for every unique service on both the server (Java) and client (LabVIEW™) side. This was one of the main points we wanted to address and one of the incentives for the project: introducing common virtual classes in the software stack that easily could be overridden for specific services. At the same time, we wanted the new service to be capable of adapting to all future changes, but keeping the existing clients backwards compatible and uninterrupted.

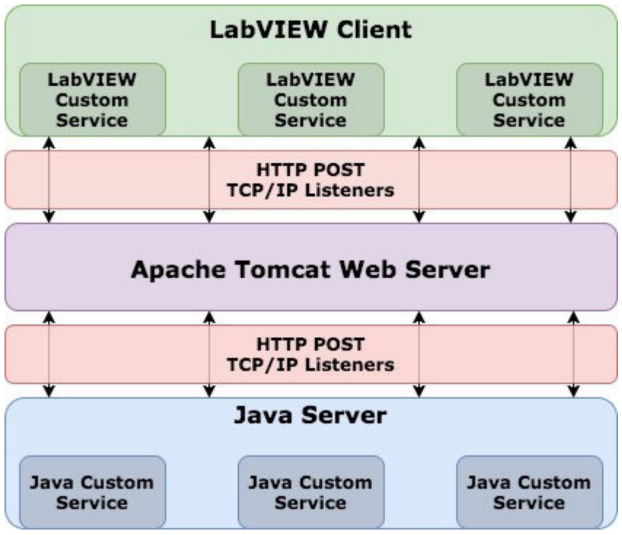

Figure 3: Apache Tomcat Web Server Implementation Architecture

Broker versus Peer to Peer Communication

Depending on the requirements, it is possible to choose either a broker or broker-less approach. The choice resides mainly on four features: scalability, maintenance, availability and management.

In a broker-less, peer to peer based system, the communication is typically implemented independently, case by case, by the developer. The implementation can be based on a common class and communication technology, but typically, as time goes by, these common components gets adapted to serve new needs in each unique component.

Also, if you need to inspect or handle multiple peers/connections simultaneously (scalability), have them join servers or access services and at the same time (load balancing) risk network outage and infrastructure downtime (availability), a message storage broker approach is more suited due to its compartmentalized nature.

Issues with consumer response time also benefit from the ability to distinguish between network error and messages lost in transit. Another important aspect is the management of the system. A broker-based approach provides centralized information about connected clients, which improves the scalability of your system.

Even though peer to peer models often are simpler to implement, their complexity grows when factoring in maintenance, availability and management. For medium to large sized applications it is almost always more advantageous to make use of brokers instead of implementing features individually.

RabbitMQ versus Apache Kafka

Table 1. presents a comparison between the RabbitMQ broker and Kafka. Kafka claims to have a five times faster speed of processing events than RabbitMQ, but it lacks the advantages of high availability and message acknowledgement. This might prove to be a liability when valuing data integrity and trying to prevent loss of data in critical applications [2] [11].

RabbitMQ also supports additional AMQP communication channels through its vast amounts of plugins, whereas Kafka only supports exchanges. While messages in Kafka stream are ordered, the delivery (ordered) is only guaranteed if they are published on a single channel, passed through one exchange and queue and received by one channel [2] [11].

The advantage of using Kafka over RabbitMQ for the RADE Services boils down to mainly the speed of receiving and sending messages. RabbitMQ shows better support on dedicated protocols, it provides acknowledgements and high availability, which improves stability and overall security in the architecture [2] [11].

Table 1: RabbitMQ and Kafka Comparison

	RabbitMQ	**Kafka**	**Custom TomCat**
Type	Broker-centric	Producer-centric	P2P
Speed	120k events/sec	280-350k events/sec	124k events/sec
HA	Yes	No	No
ACK	Yes	No	
AMQP	Exchange, binding queues	Exchange (no queue)	No
Delivery	Ordered	Ordered (with limitations)	FIFO

In addition to the above-mentioned features, RabbitMQ also provides other benefits that make it good choice compared to other message brokers: an open, well defined and community supported AMQP standard, Erlang-based implementation that allows simple clustering and scalability. It is also more reliable and crash resistant than Apache Kafka [11].

ARCHITECTURE

The new system architecture consists of four main layers: The service layer, the application Layer, the entity layer and the data layer (See Figure 4).

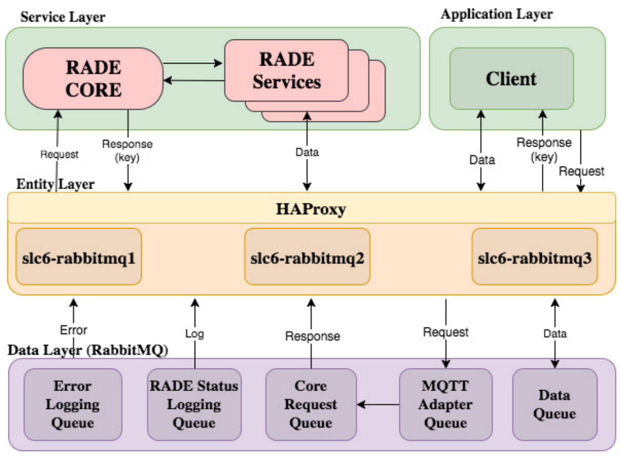

Figure 4: Project architecture.

The RADE CORE Service is a generic Java based virtual class that extends all the communication layers needed to connect to the CERN technical infrastructure (such as JAPC, CALS, ALARM etc.). Clients (LabVIEW™ and web) communicate with the RADE CORE service via the "entity layer". The entity layer proxies' data to a free broker using a designated routing key. The routing key ensures that the messages sent are properly routed to the intended subscriber/listener and guarantees delivery in a well-balanced cluster. Once a communication link has been established, a dedicated data queue is created in the data layer, facilitating communication between the requested service and its client, via the routing key [2].

The Data Layer contains five main queues, defined within the RabbitMQ management system:

- Error queue
- RADE Server status log queue
- Core request queue
- MQTT adapter queue
- Data queue

The MQTT Adapter Queue, created by the MQTT plugin is automatically generated when a subscription is initiated.

The current "entity layer" is comprised of three RabbitMQ servers running Scientific Linux 6. The servers are set up to be a single cluster, and the load balancer automatically selects which of the three services to use [2].

TESTBED

The current testbed is configured to run on two virtual machines working as a cluster (based on RedHats OpenStack) [12]. In this configuration clients can connect to any node in the cluster and connect to all or any existing queue. All exchanges, queues, permissions and virtual hosts are mirrored across the nodes. This configuration makes use of the High Availability Proxy (HAProxy), providing us with load balancing [13].

To communicate with LabVIEW™ and Web Clients, the following plugins were installed:

- rabbitmq_mqtt plugin for MQTT communication
- rabbitmq_management plugin for HTTP-based management

All data exchange queues, interfaces, ports and service names were manually configured in the dedicated RabbitMQ and plugin configuration files.

INTERFACES

In this project, we mainly distinguish between two types of service interfaces: The RADE Java Services and LabVIEW™ MQTT clients. What is most important in this case is the broker's support for various protocols. Apart from AMQP used in JAVA services, the chosen broker needs to provide support for the MQTT protocol that is used in the LabVIEW™ Client.

Workflow

In the test system, all clients are connected and send their requests to a dedicated broker cluster. Through the Broker the request is sent to an appropriate service that is executed depending on the request message payload. The requested information is sent back to the broker, and then relayed to the requesting client. The simplified workflow is shown on the Figure 5.

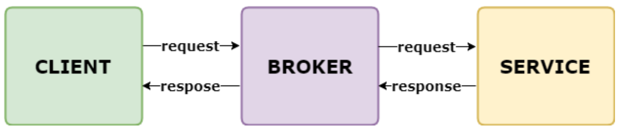

Figure 5: Workflow

LabVIEW™ Client

The choice of MQTT protocol assures that the developed applications are platform independent, which was not possible using the AMQP protocol. All available libraries using AMQP protocols are based on .NET technology which is not supported on Linux systems. The MQTT library used in our applications is based on pure TCP/IP protocol, which allows them to be executed on every platform.

Java Interface

The RADE Java Services communicates with RabbitMQ using a dedicated server library provided by the official RabbitMQ stack.

We implemented a wrapper for this library, called RADE CORE. RADE CORE functions as the main interface to the RADE Services. It features access control, resource routing and monitoring. Through its virtual and scalable implementation, one can run several instances, simultaneously, on the same or different servers. All the instances will cross communicate using the RabbitMQ broker, ensuring the data integrity being kept, and at the same time making it possible to launch multiple instances of the same service with different environment configurations.

All the RADE JAVA Services are working as plugin based extensions of RADE CORE. This makes it possible to execute specific actions for each RADE Service.

CROSS PLATFORM COMMUNICATION

The RabbitMQ messages between LabVIEW™ and JAVA was serialized using JSON strings or LabVIEW™ variants (binary) [14]. JSON was chosen because of its popularity and wide spread, facilitating inter-language communication, while LabVIEW™ variants was used to ensure backwards compatibility with the existing services.

When launched, the RADE CORE initializes a communication channel with the broker using a native AMQP protocol. This gives the core service all the information to track the connections, register queue names, and all the routing keys in use.

From the client side (typically LabVIEW™), the communication is mainly using the MQTT protocol. Information is sent using a service-dedicated routing key to the broker. This routing key ensures that the request arrives at the right services, and it makes sure that the data exchanged is kept intact [10].

When a client connects, The MQTT subscription queue is automatically created. Our broker, working as a cluster, processes the request and passes the message through to RADE CORE interface. Here, depending on the request, the dedicated plugin is launched.

Once a communication channel has been established in the RabbitMQ cluster, the requested data is sent to the broker and then to the subscribed client.

Once the data exchange is done (ranging from milliseconds to days) the client sends a request to close the service.

Finally, when the service has shut down, the RabbitMQ broker shuts down the communication channel and deletes the message queue.

VALIDATION

The RADE Service implementation of the RabbitMQ broker was tested in terms of speed, reliability, performance and throughput and compared with the older Apache Tomcat based web service implementation. All the tests were conducted using built in Linux tools such as nload (network traffic), top and the regular system monitor. We observed no significant difference between speeds while sending small sized (< 1kB) messages, but it was observed that on larger messages (>1MB) the RabbitMQ cluster would lag behind slightly compared to Apache Tomcat custom implementation. This however could be compensated by reducing the package size and deploying several clusters. In addition, the RabbitMQ solution provides authentication and a guarantee of message delivery which the custom server doesn't have, and for stream like connections (video feeds, Beam Position Monitors, etc.), Apache Kafka is considered being deployed.

CONCLUSION

Introducing RabbitMQ broker with AMQP and MQTT extension makes it possible to improve scalability, redundancy, performance and development flexibility for our applications. Lightweight, open source protocols provide cross platform and cross language communication.

Even though the broker-based implementation is less performant in terms of throughput compared to the previous implementation, a cluster based backend such as RabbitMQ gives us a safer and more robust architecture, which assures that no messages will be lost in case of node failures or minor interruptions in the network. All of the mentioned features, as well as increase in flexibility, help us to move forward towards more standardized and efficient applications with smaller footprints.

FUTURE PLANS

As a future step, it is planned to create an access control layer, where clients will connect using RabbitMQ and separate RADE Servers for each individual service.

Access control, depending on the type of the request, will select the appropriate communication type, for example RabbitMQ or Kafka, and forward the request to the specific RADE CORE.

REFERENCES

[1] O. Ø. Andreassen et al. "The LabVIEWTM RADE framework distributed architecture", ICALEPCS 2011, Grenoble, France, WEMAU003.
[2] RabbitMQ website: http://www.rabbitmq.com.
[3] Z. Zaharieva, "Database foundation for the configuration management of the CERN accelerator controls systems", ICALEPS2011, Grenoble, France, MOMAU004.
[4] K. Kostro,"The control middleware (CMW) at CERN status and usage", ICALEPCS2003, Gyeongju, Korea, WE201.
[5] CERN Information Technology Department website: http://information-technology.web.cern.ch/services.
[6] InCa website: https://espace.cern.ch/be-dep-workspace/op/ps/InCA%20PS/InCA%20for%20Dummies.aspx
[7] DIM website: http://dim.web.cern.ch/dim
[8] Apache Tomcat website: http://tomcat.apache.org/
[9] AMQP website: https://www.amqp.org/.
[10] MQTT website: http://mqtt.org/.
[11] Apache Kafka website: https://kafka.apache.org/.
[12] OpenStack website: https://www.openstack.org/
[13] HAProxy website: http://www.haproxy.org/
[14] JSON website: http://www.json.org/