The eBADGE Message Bus – First Intermediate Version

Deliverable: **D3.2.1**

Of project: **eBADGE**

Grant Agreement no. **318050**

Project Full Title:

Development of Novel ICT tools for integrated Balancing Market Enabling Aggregated Demand Response and Distributed Generation Capacity

Project Duration: **3 years**

Project start date: **October 1st, 2012**

SP1 – Cooperation

Collaborative project

Small or medium-scale focused research project

Due date of deliverable:	Month 12 (30. 9. 2013)
Workpackage:	WP3
WP Coordinator:	XLAB
Authors:	Daniel Vladušič, Staš Strozak, Marjan Šterk (XLAB) Radovan Sernec, Marko Rizvič (TS) Ingo Sauer (Sauper)

Dissemination level		
PU	Public	**X**
RP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

Table of Contents

Executive Summary

This document describes the process that produced the first version of the eBADGE message bus, which will be used for communication between all the stakeholders in the pilot project. Excellent security and reliability, easy set-up and maintenance, low overhead and high performance can be obtained by using a simple messaging proxy as opposed to direct communication between the stakeholders on one hand or a full enterprise service bus on the other, both of which lack one or more of the above requirements.

We enumerate numerous widely used messaging protocols and their implementations and give the requirements and key performance indicators with which to judge them. After a careful evaluation we have selected RabbitMQ, an implementation of AMQP, as the most appropriate.

We have developed a library that simplifies sending and receiving eBADGE data standard messages as well as mapping them from and to Python objects. This open source library and a default RabbitMQ installation form the first version of the eBADGE message bus.

1 Introduction

The aim of task 3.2 is to build a robust and high performance message bus between all the stakeholders in the pilot, i.e. resources (loads, distributed generators – through home energy hubs (HEHs)), VPPs, DSOs, TSOs and possibly others. The message bus should be built using off-the-shelf open source components and the message payload will be the eBADGE standard messages, as defined in [eBADGE D3.1.1, 2013].

The main result is a software prototype – a library that maps eBADGE data standard messages to Python objects and simplifies sending and receiving these messages. This document is thus merely a report on which technologies were chosen and why, how this first version was implemented and how to use it.

1.1 Approach

We started by analyzing the possible ways to connect the stakeholders, either through direct stakeholder-to-stakeholder connection or through some kind of a proxy. We found out that the latter option is preferable. Suitable message bus implementations were analyzed. It was decided that for this first version RabbitMQ can be used as-is, with no modifications in messaging per se. However, in order for the message bus to simplify as much as possible the sending and receiving of the eBADGE standard messages, an object-message mapping library was developed.

1.2 Outline of This Report

The next section discusses the communication architectures with and without a central proxy and our arguments for using a simple message broker. Section 3 contains the message bus requirements and a comparison of existing message bus technologies. Section 4 describes our actual implementation, that is, RabbitMQ with a more specific library on top of it.

2 Communication Architecture

The basic architectural choice is how to connect all the stakeholders in the domain. As we are talking about multiple stakeholders in each of the countries involved and also connections between the different countries (either through one proxy or in a more chaotic way), there are basically two possibilities to make the connections:

- Through a direct stakeholder-to-stakeholder connection,
- Through some kind of a proxy.

As the first option requires a lot of work for each of the stakeholders (either users, either providers, brokers, etc.), the option with a proxy seems much better. Technically, it means that the proxy is the entity that defines the API and thus everyone else will adapt to it. Finally, such model will require the stakeholders to make only one translation from their native data/API calls.

The main difference between the two possibilities is thus the uniformity of the chosen solution, which manifests as the standard (API) each of the stakeholders has to adapt to vs. full-graph of possible solutions.

Let us summarize the requirements that influence this decision:

- Fully defined API for each of the stakeholders,
- Network connections established only from stakeholders to proxy, not in the other direction,
- State-of-the-art security layer,
- Redundancy/failover in case of failure of (part of) proxy.

We need to stress that we are discussing only the transport level APIs/protocols between the stakeholders, i.e. the “HOW” and not the WHICH DATA and WHAT does the data mean.

2.1 Conceptual presentation of the broker

Figure 1 shows one possible implementation with all the stakeholders being able to communicate to each other. The pattern assumes that the communication standard (the message payload) is standardised for the sender/producer and receiver/consumer so that they can indeed communicate and that the messaging is being performed over the internet. Other (more technical) details are currently disregarded. Also, the communication between different stakeholders in the figure is conceptually possible while in reality it's seldom performed as there's no practical need for it. This fact has also been disregarded. The Figure 1 thus simply shows what can technically be achieved.

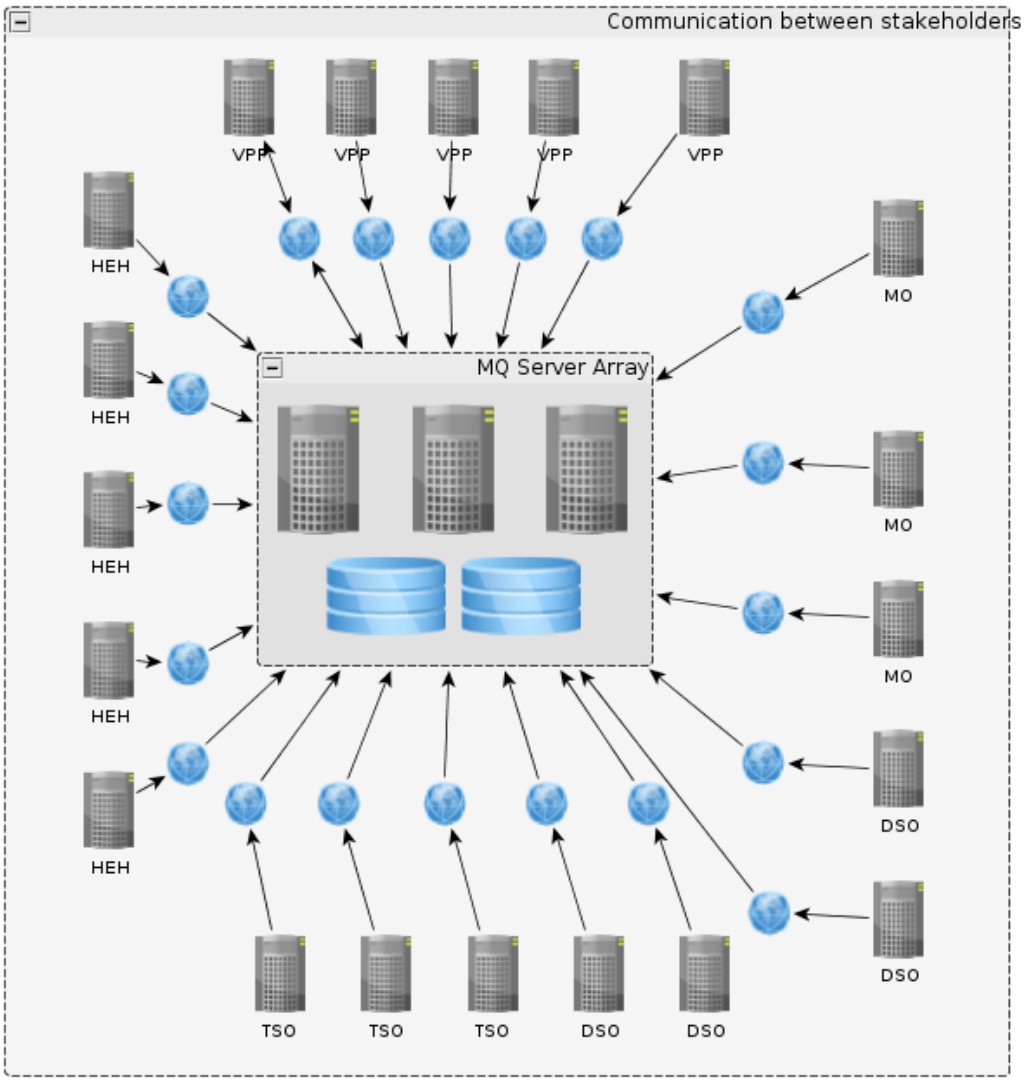

Figure 1: A possible communication pattern between all of the stakeholders

2.2 Choice of a "Messaging" System

The review of the possible messaging systems shows two main possibilities:

- Enterprise Service Bus (ESB),
- Messaging Queuing Service (MQ).

For eBADGE, we choose the MQ. While the ESB and MQ may seem identical, it is generally accepted that ESBs offer more functionality than MQs. However, the MQs are easier to set-up, are viewed as more versatile and lighter-weight than ESBs. As an illustration of this fact, please see the typical ESB stack in Figure 2. There are a lot of layers that, in our case, would only complicate the communication between the stakeholders in eBADGE. However, should there be a need to introduce such an ESB, the move from MQ to ESB is rather trivial in terms of setting up the

communication patterns and changing the already developed eBADGE components. E.g., for Apache Service Mix, the internal messaging system is actually Apache ActiveMQ (please see: https://servicemix.apache.org/). Thus, eBADGE is adopting a simpler solution that can be easily upgraded into a more complicated one, if there is ever a need for that.

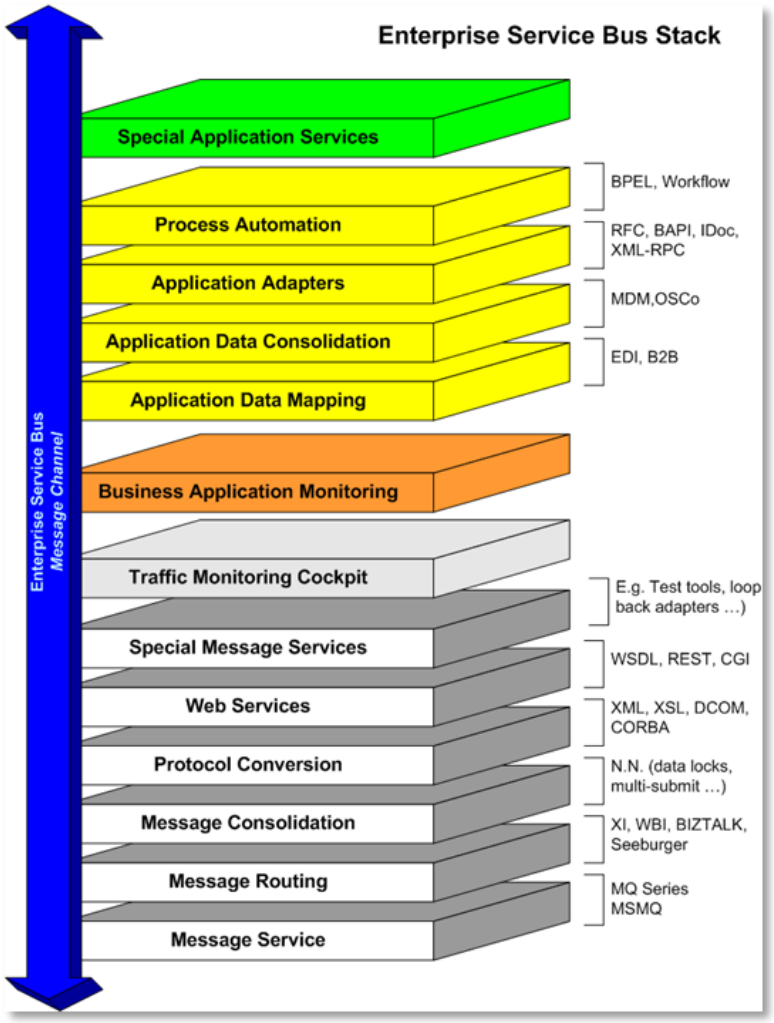

Figure 2: A typical ESB stack[1]

[1] Source: https://en.wikipedia.org/wiki/File:ESB_Component_Hive.png

3 Selection of Message Bus

Table 1 shows different message queues – the list is however incomplete, as there is a plethora of different MQs available. The only exception is Redis, which is a distributed key/value store rather than MQ, but can be used to achieve similar functionality. The reason for its inclusion its current popularity and also the fact that people have used it for such functionality with success. Also, ZeroMQ is more similar to Unix sockets rather than a fully-featured message bus.

Name	**Website**	**License**	**Implemented standards**
ZeroMQ	http://www.zeromq.org/	GPL/LGPL	/
RabbitMQ	http://www.rabbitmq.com/	MPL	AMQP 0.9.1[2]
IronMQ	http://www.iron.io/	Commercial	/
WebSphere MQ	http://www-03.ibm.com/software/products/us/en/wmq/	Commercial	/
BeansTalkD	http://kr.github.io/beanstalkd/	MIT license	/
Amazon Simple Queue Service	https://aws.amazon.com/sqs/	Commercial	/
ActiveMQ	https://activemq.apache.org/	Apache license	AMQP 1.0
OpenAMQ	http://www.openamq.org/	GPLv3 server, BSD client	AMQP 0.9.1
Apollo 1.6	http://activemq.apache.org/apollo/	Apache 2.0	AMQP 1.0
Redis	http://redis.io/	BSD license	/

[2] Please note that AMQP 0.9.1 and AMQP 1.0 are separate messaging standards, rather than the former being just a draft version of the latter.

Apache Vysper	https://mina.apache.org/vysper-project/index.html	Apache license 2.0	XMPP[3]
OpenFire	http://www.igniterealtime.org/projects/openfire	Apache License	XMPP

Table 1: A short list of different MQs and similar appropriate technologies.

When considering what eBADGE needs in terms of MQ, we outline the following properties/requirements that the target system shall have:

- Feature-completeness: we want to have as many possibilities as possible. Mainly, we are interested in different bridges with technologies (e.g., bridge towards XMPP protocol)
- Message-persistence: we do not want simply high-speed, but transient messaging – we require message persistence.
- Language bindings: as we think about future use of the eBADGE message queues, we want to have as many as possible language bindings.
- Robustness and stability, as proven through a wide base of deployments and a large community. This also ensures that it will not go out of fashion and out of support prematurely.
- Interoperability (cross platforms – support for different operating systems, programming languages)
- Based on open standard and preferrably open source implementation
- Efficiency (depending on the message payload standards)
- High availability
- Security: message encryption, access control/authorization, distribution of encryption keys
- Availability of software, permissive licensing model

Having the above in mind, we collated a list of further requirements and key performance indicators (KPIs) that need to be followed during the selection of the appropriate message bus solution and its operation:

- Non-functional requirements:
    - security

[3] A full list of XMPP servers can be found at: http://xmpp.org/xmpp-software/servers/.

- availability [percent downtime]
- reliability [MTBF]
- code size as applied on embedded designs (ARM)
- efficiency
  - traffic overhead per message
  - message delay
  - throughput for short messages
  - throughput for long messages
  - CPU usage with high throughput on embedded designs

- Functional requirements:
  - persistence [Messages are not lost even when MQ crashes]
  - interoperability

Additional information about the MQs and an insight into the performance of different MQs can be found in [Mihailescu, 2013; Salvan, 2013].

3.1 RabbitMQ as the eBADGE Message Bus

Based on the feature list above, we have chosen RabbitMQ, due to its message persistence, open-source nature, plethora of possible extensions and good language bindings. Finally, RabbitMQ is not directly connected with Java (as, e.g., ActiveMQ is), rather it's written in Erlang. Currently, we're trying to avoid Java for the following reasons:

- for future long-term developments, it is unclear how Oracle will proceed with Java SE. Older versions already requires subscription to receive critical updates.
- the HEH most probably does not qualify as a “general computing device”, thus distributing Oracle Java installed on the HEH requires negotiating a license.
- OpenJDK, while avoiding the above problems, is significantly slower and somewhat more bug-prone.

We have also tested RabbitMQ on the Raspberry Pi platform that will be used for Home Energy Hubs. We have confirmed that it works out of the box, requirying installation of approximately 15 MB of packages. The performance was also found to be more than adequate for sending the foreseen rate of a few messages per minute.

4 The eBADGE Message Bus – Implementation, First Version

Having selected RabbitMQ as the basis, we have to:

- extend it with any unsupported but required functionalities,
- fix any critical bugs that are not fixed in time by its developers or other community members,
- provide extensions to simplify its usage in the context of eBADGE.

We have thus far not found any critical bugs. On the other hand, we have found one missing feature – end-to-end encryption.

RabbitMQ supports encryption of messages in transit, including secure exchange of keys, certificates etc. However, since in general its main paradigm is many-to-many communication, it decrypts each message on arrival into the RabbitMQ server and encrypts it again before dispatching it to the consumer, so that each stakeholder must only exchange keys with the server rather than with all the consumers of its messages and all the producers of the messages it wants to read.

In eBADGE, all (or almost all) messages will in fact have a single sender and a single receiver; thus it is feasible to implement end-to-end encryption. At the very least, all messages at the market level must be encrypted and signed in order for a malicious RabbitMQ server administrator not to be able to read nor forge messages critical for grid security. We plan to add encryption on top of RabbitMQ in the second or third version of the message bus.

Reliability of the messaging server is also a critical requirement. RabbitMQ supports setting up multiple, redundant servers to achieve state-of-the art reliability. This has already been tested in an isolated test. In the later versions we will enable redundancy in the default installation of eBADGE message bus.

4.1 Mapping RabbitMQ Messages to Object

To simplify the development of eBADGE software components, this first version of the message bus already includes a reference implementation of the eBADGE data standard. This is implemented as a Python library that maps each message type specified in the eBADGE data standard to a corresponding Python class, with full json serialization/deserialization.

The basic json (de)serialization methods included in Python had to be customized to achieve full compliance with the data standard. For example, Python *datetime* objects typically do not have a specified time-zone, the standard forbids this to avoid confusion. Also, certain field names in eBADGE messages are reserved names in Python, also requiring special processing.

4.2 Sending and Receiving eBADGE Messages

The message bus also offers a simplified API for setting up the connection to RabbitMQ server and for the two basic communication operations, i.e. sending a single message and waiting for a specific message in order to act on it. As the below code excerpt illustrates, usage is extremely simple; on the other hand, the API is transparent in that all the RabbitMQ objects and features are accessible through the eBADGE message bus API.

```
import ebadge_msg
import sys

upConnector = ebadge_msg.Connector(sys.argv[1], sys.argv[2] + '-up')
downConnector = ebadge_msg.Connector(sys.argv[1], sys.argv[2] + '-down')

class Consumer(ebadge_msg.AbstractConsumer):
    def on_get_load_report(self, request):
        report = ebadge_msg.LoadReport(
                    request.from_, request.to,
                    request.resolution, request.device,
                    [0.12, 0.17, 0.33, 0])
        upConnector.basic_publish(report)

downConnector.start_consuming(Consumer())
```

The code first creates two eBADGE MB connectors, one for upward and one for downward communication. It then implements a message consumer that on receipt of message type *get_load_report* will create a LoadReport object (a mapping of *load_report* message type), fill it with hard-coded contents and send it. Finally, the code starts this message consumer.

4.3 Obtaining the Sources

The reference Python implementation of the eBADGE data standard can be downloaded from https://dev.xlab.si/ebadge/ebadge-svn/wp3/message-bus/releases/1.0/. It can be downloaded file-by-file or checked-out with the svn client:

svn co https://dev.xlab.si/ebadge/ebadge-svn/wp3/message-bus/releases/1.0/

No username or password is required for read-only access.

5 Conclusions

We have enumerated direct communication, a simple message broker, and a fully-fledged enterprise service bus as the possible communication architectures. We argue that a simple message broker is the optimal choice for eBADGE, both for communication with the Home Energy Hubs as well as communications on the balancing market.

We have evaluated the most widely used message bus standards and implementations and defined the requirements and key performance indicators that the selected implementation must meet. On this basis we selected RabbitMQ, an implementation of AMQP 0.9.1, for the first version of the eBADGE message bus.

We have performed the first round of tests on RabbitMQ and found that currently the only missing feature is end-to-end encryption (instead, RabbitMQ offers source-to-broker and broker-to-destination encryption, which is not enough for our purposes). This will be implemented later, thus at this stage the eBADGE messaging channel is RabbitMQ as-is, without any extensions.

We have provided a reference implementation of the eBADGE data standard as a Python library, which provides a two-way mapping between eBADGE messages and Python objects and a simplified AMQP-like API for sending and receiving the messages. It is publicly available under a free software license. The first version of the eBADGE message bus thus consists of this library and a default RabbitMQ installation.

References

eBADGE D3.1.1, 2013: The eBADGE Data Standard Report – First Version, eBADGE project deliverable D3.1.1, September 2013.

Mihailescu, 2013: Adina Mihailescu, Messaging Systems – How to make the right choice? http://rivierarb.fr/presentations/messaging-systems/ (accessed on 17th of May 2013)

Salvan, 2013: Muriel Salvan, A quick message queue benchmark, http://x-aeon.com/wp/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/ (accessed on 17th of May 2013)