The eBADGE Message Bus
Final Version

Deliverable report

Document Information

Programme	FP7 – Cooperation / ICT
Project acronym	eBADGE
Project Full Title	Development of Novel ICT tools for integrated Balancing Market Enabling Aggregated Demand Response and Distributed Generation Capacity
Grant agreement number	318050
Number of the Deliverable	D3.2.3
WP/Task related	WP3 / T3.2
Type (distribution level)[1]	PU
Due date of deliverable	30.09.2015 (Month 36)
Date of delivery	30.09.2015
Status and Version	Final, v1.0
Number of pages	30 pages
Document Responsible	Marjan Šterk - XLAB
Author(s)	Daniel Vladušič, Staš Strozak, Marjan Šterk- XLAB Radovan Sernec, Marko Rizvič, Tomaž Lovrenčič - TS Ingo Sauer, Armin Fuchs, Andreas Flür - EUDT
Reviewers	Andraž Andolšek - CG

1

PU	Public
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)

Issue record

Version	Date	Author(s)	Notes	Status
1.1	27.01.2016	Marjan Šterk, XLAB	Updated link to software repository	Final
1.0	24.09.2015	Marjan Šterk, XLAB	Integrated suggestions from internal review	Final
0.4	04.09.2015	Marjan Šterk, XLAB	Updated conclusions, introduction, executive summary, glossary	Draft
0.3	04.09.2015	Marjan Šterk, XLAB	Added specifications for high-availability and summary of comparison to XMPP	Draft
0.2	10.08.2015	Marjan Šterk, XLAB	More details on security setup in Chapter 5. Minor updates throughout.	Draft
0.1	07.08.2015	Marjan Šterk, XLAB	Renamed D3.2.2 to D3.2.3, updated administrative parts	Draft

NOTICE
The research leading to the results presented in the document has received funding from the European Community's Seventh Framework Programme under grant agreement n. 318050.

Table of Contents

Table of Figures

Table of Tables

Table of acronyms

Acronym	Meaning
ACER	Agency for the Cooperation of Energy Regulators
ACK	Acknowledgement
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
BSP	Balancing Service Provider
CPU	Central Processing Unit
DSO	Distribution System Operator
ESB	Enterprise Service Bus
HEH	Home Energy Hub, a metering and controlling device developed in eBADGE WP4
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
ICT	Information and Communication Technologies
IP	Internet Protocol
JSON	JavaScript Object Notation
LTE	Long Term Evolution (4G mobile technology)
MB	Message Bus
MQ	Message Queue
MQTT	MQ (Message Queuing) Telemetry Transport
OpenADR	Open Automated Demand Response, a demand-response messaging standard
PKI	Public-Key Infrastructure
REST	Representational State Transfer
STOMP	Simple (or Streaming) Text Oriented Messaging Protocol
TCP	Transfer Control Protocol
TS	Telekom Slovenije
TSO	Transmission System Operator
TTL	time-to-live
VPP	Virtual Power Plant
WP	Work Package
VRRP	Virtual Router Redundancy Protocol
WS	Web Service
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol

Glossary

Term	Meaning
.NET	a software framework and run-time environment for multiple programming languages (e.g. C#, Visual Basic.NET), developed by Microsoft
ActiveMQ	a message bus implementation by Apache
C	a low-level programming language
C++	an object-oriented programming language based on C
credentials	a proof of identity of a user or component of an ICT system; for example, username+password
binding	a routing rule in AMQP specifying which messages should be routed where
data model	a specification of data types (objects) and the fields they contain; in this document refers to the eBADGE data model [1]
durable queue	a queue in AMQP that does not disappear when the message bus server is restarted
eavesdropping	listening on messages intended for somebody else
end-to-end encryption	encryption of messages at the sender side and decryption on the receiver side, such that no intermediaries can see unencrypted contents
enterprise service bus	a software architecture and corresponding multi-layer software stack for communication between mutually interacting software applications in a service-oriented architecture
Erlang	a parallel programming language often used in telecommunications
exchange	a concept in AMQP where messages are sent to, analogous to a mail drop box
failover	the act where running instances of a replicated entity take over from failed instance
fibre-to-the-home	optical fibre that runs directly to a home or office, typically used for Internet access
firewall	a (hardware or software) network device that blocks disallowed communication, typically based on IP addresses and port numbers
heartbeat	messages that are periodically sent in order to let the receiver know that the sender is still there
high availability	a property of an ICT system that it is available (almost) always, even at times of failure of part(s) of the system
Java	a programming language and corresponding run-time environment developed by Oracle (originally by Sun Microsystems)
load balancing	the act of distributing requests to multiple servers so that the latter are as equally loaded as possible
market model	a mathematical model of balancing energy market, being developed in eBADGE WP2
market simulator	a software simulator of the market model, being developed in eBADGE WP2
message bus	a generic name for a messaging proxy, the required server and client software, typically also capable of one-to-many communication
message header	the part of a message that contains the message metadata and typically precedes the payload
message payload	the part of a message that contains the actual data and typically follows the header
message queue	a data structure within a messaging proxy where messages are buffered until they are delivered to the destination end-point, analogous to a private incoming mail box
message server	an intermediary server that routes messages from a sending entity (end-point) to the receiving entity in the absence of direct connection
messaging proxy	an intermediary entity (typically, a server) through which one entity (end-point) can communicate with another without requiring a direct connection
nameless exchange	a default exchange in RabbitMQ with pre-defined queue bindings
Perl	a family of high-level programming languages
proxy	an intermediary entity through which one entity (end-point) can reach another entity in the absence of direct access
Python	a high-level programming language
Raspberry Pi	the type of single-board computer used in the eBADGE HEH
replication	use of multiple instances of the same entity so that if one instance fails the rest can take over

round-trip time	the time between sending a message and receiving a response
routing key	a metadata field of each AMQP message used to specify bindings
Ruby	a high-level programming language
serialization/deserialization	the process of transforming a software object to/from a stream of bytes
service delivery broker	a control layer or service that mediates access to services and resources
spoofing	forging a message, pretending it was sent by somebody else
svn	a software versioning and revision control system by the Apache Software Foundation, also known as Subversion
time-to-live	a time period after which a message, if not yet delivered, is automatically deleted
topic exchange	a type of AMQP exchange that routes messages based on their routing key using wildcard matching
ZeroMQ	a high-performance asynchronous messaging library

eBADGE Project

The 3rd Energy Package clearly boosts the development of an Integrated European balancing mechanism. In this context, ACER has in 2011 started the development of the Framework Guidelines on Electricity Balancing. It is expected from the ACER statements that Demand Response will play significant role in the future integrated balancing market allowing Virtual Power Plants, comprising Demand Response and Distributed Generation resources to compete on equal ground.

The overall objective of the eBadge project is to propose an optimal pan-European Intelligent Balancing mechanism also able to integrate Virtual Power Plant Systems by means of an integrated communication infrastructure that can assist in the management of the electricity Transmission and Distribution grids in an optimized, controlled and secure manner.

In order to achieve the above overall objective the eBadge project will have four objectives focusing on:

1. Developing the components: simulation and modelling tool; message bus; VPP data analysis, optimisation and control strategies; home energy cloud; and business models between Energy, ICT and Residential Consumers sector;
2. Integrating the above components into a single system;
3. Validating these in lab and field trials;
4. Evaluating its impact.

Project Partners

Telekom Slovenije d.d. - Slovenia
cyberGRID GmbH - Austria
Ricerca sul sistema energetico – RSE Spa - Italy
XLAB Razvoj programske opreme in svetovanje d.o.o. - Slovenia
ELES d.o.o. - Slovenia
Austrian Power Grid - Austria
Borzen, Organizator trga z električno energijo, d.o.o. - Slovenia
Elektro Ljubljana, Podjetje za distribucijo električne energije d.d. - Slovenia
Technische Universitaet Wien - Austria
Austrian Institute of Technology GmbH - Austria
EUDT Energie- u. Umweltdaten Treuhand GmbH - Austria
SAP AG - Germany
Vaasaett Ltd Ab Oy - Finland

Project webpage

http://www.ebadge-fp7.eu/

Executive Summary

The objective of this deliverable is a software prototype – a message bus for communication among all the entities in the eBADGE pilot project and potentially applicable to the smart grid marketplace in general. The final version, which is described in this report, consists of RabbitMQ, an open source AMQP 0.9.1 implementation, as the messaging proxy, recommendations on the setup of RabbitMQ resources, and a Python library that maps objects to and from the messages of the eBADGE data model. Together they provide an easy-to-use, high performance, secure, scalable and reliable message bus. The library itself is also available under an open source license.

This document is a report accompanying the prototype deliverable. It summarizes the process that produced the three versions of the eBADGE message bus and specifies where they differ. We list the requirements, including a quantitative minimum performance specification required to show the scalability of the approach to production-sized deployments of tens of thousands demand-response sites controlled by one virtual power plant. We argue that the given requirements can be met by such a straightforward messaging proxy.

We list multiple widely used messaging protocols and their implementations, performance measurements from the available literature and describe initial network overhead analysis of ActiveMQ and RabbitMQ. We cite the comparison to XMPP, as used by the OpenADR 2.0 standard, that was done elsewhere in the project and that also showed certain advantages of RabbitMQ over XMPP. After these evaluations we have thus selected RabbitMQ as one that fits well within the eBADGE project objectives and constraints. We report on the specific implementation, configuration and naming conventions that ensure proper interoperability, security, reliability and scalability for the eBADGE pilot and also allows elaborate message routing within the receiving entity, thus easing integration with existing ICT systems.

1. Introduction

The present deliverable report is a part of WP3 “Development of VPP as a balancing asset” and in particular its task 3.2. “Analysis, Design and Implementation of the eBADGE Message Bus”. 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 messages, as defined in [1].

This version brings another update of the prototype – the library that maps eBADGE data model messages to Python objects and simplifies sending and receiving these messages over RabbitMQ. The update was necessary due to further changes in the data model. Furthermore, this version contains some changes to the recommended setup of the RabbitMQ resources such as queues and exchanges, as well as to the security aspects of the setup. A description of a high-availability RabbitMQ cluster setup is given later on. A short illustration of how RabbitMQ works is also provided to make this setup specification self-contained. Finally, this document includes the report on which technologies were chosen and why, which was already given in [2].

The main changes to this report include the addition of XMPP as a possible candidate and expansion of the security-related conventions in Section 5.3. Detailed descriptions of the configuration and deployment of RabbitMQ clusters and related services that ensure high availability and scalability were added to Section 6.

1.1 Approach

Requirements were collected from and in cooperation with both WP4, where the HEH prototype and the business models that will aggregate them into VPPs are being developed, and WP2, where the market models were analysed and a market simulator was developed. We particularly focused on the HEH-level requirements for two reasons. First, the project results at the HEH-level are much more likely to be directly re-used in commercial products. Second, in order to demonstrate the scalability of the approach to production-sized VPPs, the performance requirements at the HEH level are much higher than at the market level.

We continued by analysing the possible ways to connect the stakeholders, either through direct stakeholder-to-stakeholder connection or through some kind of a proxy/message server. We found out that the latter option is preferable in order to meet the project goals within the foreseen development effort. Accordingly, multiple message bus technologies were evaluated both by consulting the literature and by conducting hands-on analysis using actual test messages as defined by the eBADGE data model.

Once a message bus technology was chosen, initial testing against the requirements was performed and no significant problems were discovered. To ease the development in the work packages that will use the message bus, a Python library was developed on top of it whose goal is to remove the need for e.g. a HEH programmer to learn all the details of the message bus and data model. Finally, small stand-alone mock-up applications were developed that can simulate a HEH during development and testing of VPP software, and vice versa.

1.2 Relation to the Rest of the Project

This message bus prototype provides a way to exchange messages between the entities in the eBADGE pilot. The requirements for the message bus are influenced by and result from the data model developed in the same work package, the home energy hub/cloud specifications developed in WP4, and, to a lesser extent, the market model and simulator from WP2. The message bus is used by the home energy hub and the server-side software deployed in the eBADGE pilot project, thus representing an integral part of the pilot integration in WP5.

In contrast to the previous version of the message bus, which was only tested in a laboratory setting at the time of writing of the respective report [2], the version described here has been deployed, field-tested, and validated in the pilot in WP4.

Finally, in task T3.3 an extensive experimental as well as theoretical comparison of the eBADGE message bus to the most relevant related standard, the OpenADR 2.0, was performed. The message bus is compared to both communication protocols of OpenADR 2.0., i.e. XMPP and HTTPS [3].

1.3 Outline of This Report

The next section lists the requirements, which are followed by a section on top-level architecture, i.e. a discussion of communication architectures with and without a central proxy. Section 3 argues why a message bus is needed at all, in contrast to direct stakeholder-to-stakeholder communication. Section 4 compares the four message bus implementations that were evaluated in our initial round of testing. Sections 5 and 6 describe the actual implementation of the first version of the eBADGE message bus, that is, RabbitMQ with specific naming conventions and other setup recommendations with an open-source Python library on top of it.

2. Requirements

As is described in [1], the communication in the eBADGE pilot can be divided into two isolated levels:

- home energy hub (HEH) level, where the HEHs and other field equipment, HEH operators, VPPs, and other business entities (e.g. BRPs, DSOs etc, depending on the market structure) communicate,
- market-level, where the balancing service providers (BSPs) communicate with the TSOs.

The first version of the message bus [2] supposed that the HEHs will be financed and maintained either by the home owner or by the VPP, and always directly operated by the VPP. However, the development of business models in WP4 showed that in the residential market one entity (e.g., a telecommunications provider) may decide to finance, install, and maintain the HEHs and then sell services to both the home owners and other entities, such as VPPs. The eBADGE message bus must thus support, but not be limited to, centralized HEH control and measurement storage provided as a service to other entities.

The HEH-level is both more demanding than the market level in terms of message volume and more likely to be directly adopted by products developed on the basis of the eBADGE pilot.

We have enumerated the following requirements for the communication software:

- technical requirements:
  - high performance (detailed in the next subsection),
  - two-way communication through firewalls,
  - security:
    - PKI-based authentication,
    - encryption of messages,
- suitability for eBADGE pilot:
  - ease of setup, configuration, and maintenance,
  - support for the Debian GNU/Linux and Microsoft Windows operating systems,
  - support for Python (for rapid development of prototypes),
  - support for .NET and Java (for easy integration with existing business-level software of project partners),
  - no license fees,
- sustainability:
  - open-source with commercial support available,
  - no dependence on proprietary software,
  - stability, as proven through a wide base of deployments and a large community, which also ensures that it will not go out of fashion and out of support prematurely,
  - live community with active developers on all software layers.

The technical requirements must be met so that the project is implementable at all. The second group ensures that the goals of the pilot can be met with a reasonable development effort. The third group ensures that commercial products (for example, HEH-equivalent equipment with expected lifetime comparable to that of smart meters) can re-use results of the pilot project.

The communication can be either direct between entity pairs (e.g. a HEH making HTTP requests to VPP's server) or through some kind of a communication proxy (e.g. both HEH and VPP connecting into a message bus server, which then routes the messages between them). In the latter case, there are additional requirements:

- reliability: replication with failover is required so that a failed communication proxy does not bring down the whole communication,
- encryption of messages while they are being processed/waiting within the proxy (optional).

The last item is optional for the pilot because it is only relevant when the communication proxy is controlled by a third-party, for example if the VPP owns and controls the HEHs but a telecommunications company acts as an intermediary between the HEHs and the VPP on all layers from physical link to the message bus server. If the communication proxy is controlled by HEH owners (in our case by TS) then the last requirement can be omitted.

All the other requirements have to either be offered off-the-shelf by the chosen technologies/frameworks/libraries or be easily implemented either within them or on top of them in the application layer.

2.1 Performance Requirements

The performance requirements can be further detailed as follows:

- efficient use of network bandwidth,
- CPU and memory efficiency on low-power systems when communicating with one or a few entities,
- CPU and memory efficiency on high-end systems when communicating with thousands or tens of thousands entities,
- low latency.

To quantify this:

1. a HEH must be able to send 10 load profiles and similar messages (approx. 200 bytes per message) in one second with negligible CPU and memory usage on the Raspberry Pi platform[2] used for the pilot HEH, so as to demonstrate possible down-sizing to significantly less powerful (and thus cheaper) microcontrollers,
2. a VPP must be able to receive 200.000 such messages (a few from each HEH in a VPP) per minute without saturating its server's memory, CPU, or network link,
3. a VPP must be able to send 5000 activation commands (180 bytes per command) and receive responses (100 bytes each), all within five seconds, for HEHs connected over fibre-to-the-home or LTE.

The given approximate message sizes were estimated based on the eBADGE data model [1]. Other types of messages are not expected to be a performance bottleneck.

The given message volumes were chosen to demonstrate the scalability to a production deployment; in the pilot they will, of course, be much lower. The number of expected load profiles (200.000) is the product of the number of HEHs in a potential very large VPP and an estimated average number of profiles (signals) monitored in each HEH, assuming that updates are sent once per minute. The sampling frequency can be higher, with each message carrying multiple consecutive measured values.

The number of activation commands is the ratio between balancing energy bid presented to the TSO by a VPP (several MW) and estimated demand response potentials of individual homes (on the order of 1 kW). The 5 s round-trip time is required so that a VPP can collect any rejected activations and send further activation commands to additional homes within the time frame imposed by the TSO.

On the market level the expected message volume is significantly lower, thus there are no further performance requirements foreseen at this moment.

[2] 700 MHz ARM system with 512 MB RAM

3. Communication Architecture

The basic architectural choice is how to connect all the stakeholders in the domain:

- through direct stakeholder-to-stakeholder connections, such as pure TCP sockets, ZeroMQ, or HTTP,
- through some kind of a proxy, such as a message queuing server, an enterprise service bus or a service delivery broker.

The first option requires each end point (that is, the entity/stakeholder that needs to communicate with another one) to know the exact address of all other end points it wishes to communicate with. This is a significant shortcoming for the eBADGE pilot because we cannot expect to reach stable operation of the whole pilot in the first try. A central proxy, on the other hand, is the only thing everybody needs to know and because its role is only to route the messages it is easier to keep stable. Such a central proxy can also be provided and maintained by a telecommunications company, should the energy market players wish to concentrate on their core business only.

Another significant advantage of a proxy-based architecture is that all end-points establish connections from them to the proxy and no firewall opening/re-configuration is required at the end points. This may not be true for all proxy-based technologies.

In a production roll-out, in contrast to the eBADGE pilot, more mature equipment would be used; however, the cost of any on-site maintenance would be higher as well due to the larger number of devices. Also, in many business scenarios the only cost-effective connectivity option may be to re-use existing home owner's internet connection, which is almost always firewalled. The described advantages of proxy-based communication would thus be at least as important as in the pilot.

In spite of the above, we also considered ZeroMQ because it is generally seen as extremely high-performance. However, at the time of our tests (Spring 2013) only the outdated version 2.1 was supported in the ZeroMQ .NET language bindings[3]. We thus decided to dismiss ZeroMQ because .NET support is crucial for one of the project partners. The bindings were later updated and now support ZeroMQ v4.0.5, but at that point switching to ZeroMQ would require a significant re-development effort for little actual benefit.

There will be multiple instances of the proxy for reasons of security and stability in a production environment. At the very minimum the HEH-level proxy will be separate and isolated from the market-level one. In the pilot, however, a single proxy instance can be used for all entities.

3.1 Choice of a "Proxy" System

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

- enterprise service bus (ESB),
- a message bus/messaging queuing service (MB).

An ESB is typically composed of multiple layers, as shown in Figure 1. One example of an open-source enterprise service bus is the Apache Service Mix (see https://servicemix.apache.org/).

In this eBADGE work package we only need the lower, messaging layers of the shown stack, and the rest of the project could possibly use small parts of one or two upper layers. Thus we have decided to just use that instead a full ESB. As is discussed below, one of the options is Apache ActiveMQ, which is in fact the messaging part of Apache Service Mix.

[3] http://zeromq.org/bindings:clr

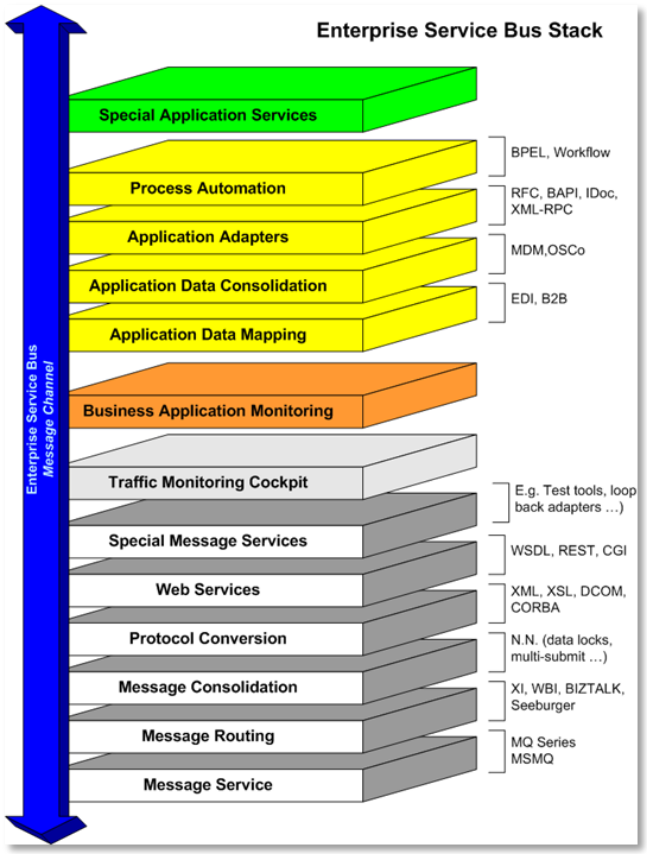

Figure 1: A typical ESB stack[4]

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

4. Selection of Message Bus

Table 1 lists a few open source message queues – the list is however incomplete, as there is a plethora of different MQs available. We have limited the evaluation to these popular choices. They are based on various messaging standards, most notably AMPQ 0.9.1 and 1.0 and XMPP. Please note that the two AMQP versions are two separate messaging standards, rather than the former being just a draft version of the latter. All of these are distributed under commercial-compatible open source licenses. The table shows the status of the AMQP implementations at the time of our initial evaluation (Spring 2013), while XMPP was added later in the course of comparison to OpenADR 2.0.

Name	Evaluated Version	Language Bindings	Supported Standards / Protocols	
			Native	Others[5]
RabbitMQ[6]	3.1.3[7]	.NET, Java, C/C++, Python, Perl, and many more	AMQP 0.9.1	STOMP, MQTT, XMPP, REST
ActiveMQ[8]	5.8	Java, C/C++, .NET, Python, and many more	OpenWire	AMQP 1.0, REST, STOMP, WS Notification, XMPP, MQTT
OpenAMQ[9]	1.4c1	C, Python[10]	AMQP 0.9.1	
Apollo[11]	1.6	Java, C/C++, Python, .NET, and more	AMQP 1.0	REST, STOMP, WS Notification, XMPP, OpenWire, MQTT
ejabberd	2.1.11	.NET, Java, C/C++, JavaScript, Pytjon, and many more	XMPP	SIP
MongooseIM	1.5.1		XMPP	WebSockets

Table 1: A list of the considered MQs.

OpenAMQ was discarded due to lack of support for .NET and other languages. At the beginning of the development of the eBADGE message bus, we decided to look deeper into two of the rest:

- RabbitMQ, which one of the partners has extensive experience with,
- ActiveMQ because of its longer presence on the market than Apollo.

4.1 Performance Comparison

The available literature suggests that, depending on the exact test, RabbitMQ is comparable or faster than ActiveMQ [4, 5]. These comparisons were done using the AMQP and STOMP protocols.

[5] Installation of additional plug-ins may be required.

[6] http://www.rabbitmq.com/

[7] RabbitMQ 3.1.3 was evaluated in the first year of project – the results of that evaluation is given in this report. The comparison to OpenADR given in [3], however, is based on the newer RabbitMQ 3.5.1.

[8] https://activemq.apache.org/

[9] http://www.openamq.org/

[10] The web site also lists Java and Ruby but the links are dead or outdated.

[11] http://activemq.apache.org/apollo/

To gain additional insight into ActiveMQ we analysed network packets with its default protocol, OpenWire, and compared it to RabbitMQ/AMQP. We took several recordings of communication for evaluation. Our network analysis software of choice here was Wireshark[12], as it records communication and also provides various options for analysis.

4.1.1. Evaluation of RabbitMQ advanced message queuing protocol (AMQP) communication

In the case of RabbitMQ, it initially looked as if the overheads would be quite high because it sends a few packets at the beginning in particular. However, the basic settings such as connection setup and negotiating the exchanges etc. should not be counted, as they are one-off costs and only sent for setting up the connection.
To test the messages and therefore communication as realistically as possible, we used the JSON messages from the eBADGE data model for the trial runs. The server was used purely as a broker and the messages were sent by two external notebooks to the system and also received again from there.

Each transmission process has three packets that are sent: first of all is *basic.publish*, then a content header and finally the content body, all of which are sent separately.

The *basic.publish* packets are relatively small, ranging between 70 and 98 bytes in size with an average of 92.5 bytes. The routing key and the name of the exchange, which are the two parts of the RabbitMQ-equivalent of an address, have at least a slight effect on the size of these packets.

The receipt process on the other hand only has one packet that is sent, called *Basic.DeliverContent-Header Content-Body*, and another packet that acknowledges receipt (called *Basic.ACK*). Communication is always single-sided, as the system waits for messages, but in the interim there is no communication whatsoever with the server. The single exception however being the heartbeats used to inform the communicating parties that they are still online and available.

The acknowledgement packets range between 75 and 87 bytes in size.
The size of the content headers is 76 to 98 bytes, with the average size for 31 packets 91.6 bytes.

Basic.DeliverContent-Header Content-Body packets are 204 to 456 bytes in size. The average size for 32 packets is 288.9 bytes.

4.1.2. Evaluation of ActiveMQ OpenWire communication

At the default settings, ActiveMQ uses the OpenWire protocol for communication in order to send messages. It first transmits information about the protocol, broker and connection on the network, followed by information that is sent once only while a connection is being established. We initially ran ActiveMQ with a limited number of messages (around 30) to gain an insight into the size of the messages and their communication. In our tests we used volatile messages rather than persistent.

ActiveMQ sends a message in a single packet, rather than a separate packet for every action. It puts a message for example in a packet called *ActiveMQTextMessage* together with all the information required, such as where the message is to go, what priority it has and naturally the message itself. This is why the messages are also larger with an average size of 479.52 bytes. Here again, the largest message is the one containing the JSON object ‘capabilities’. Note that this is a different approach from RabbitMQ, where the model implemented is separate packet per action.

The messages are forwarded to the recipients with *MessageDispatch* packets, which in turn contain all the information required for processing the messages internally. These packets are of an average size of 583.12 bytes for 25 packets sent.

The *Acknowledge* packets are larger than the RabbitMQ ones. Their average size is 325.9 bytes but again they include all the information on the destination.

[12] http://www.wireshark.org/

4.2 Final Choice

After working on the prototype for some time until we achieved almost the same application-level functionality with ActiveMQ as with RabbitMQ, we did not find a significant advantage of one over the other in the context of the eBADGE pilot project. To shorten the learning curve (one of the partners has extensive experience with using RabbitMQ) we selected RabbitMQ for the first version of the message bus.

The extensive comparison with OpenADR 2.0 brought XMPP back into consideration. As results reported in [3] show, RabbitMQ is somewhat better as far as security and reliability is concerned. The same report also shows a big performance advantage of eBADGE message bus (that is, eBADGE data model JSON messages over RabbitMQ) compared to OpenADR 2.0 over XMPP; however, this is mostly a consequence of the larger XML messages in OpenADR 2.0 and the fact that the latter uses a synchronous request-response communication model while the eBADGE data model uses asynchronous messages.

5. RabbitMQ Resource Set-up in eBADGE Message Bus

AMQP 0.9.1, which RabbitMQ implements, is a generic messaging protocol that can be used to implement a plethora of different communication patterns. Entities that wish to communicate must reach a consensus on when to use which pattern, how to name the respective AMQP resources (such as message queues), and what permissions each entity should have on these resources.

The eBADGE message bus specifies all these details. Any entities that implement the eBADGE data model and uses the message bus as specified here will thus be able to communicate with each other. This specification is a result of experience gained in the process of deployment and operation of the previous two versions of the message bus into the eBADGE pilot and has also itself been deployed and thus validated in the pilot.

5.1 Introduction to AMQP/RabbitMQ

A short introduction to AMQP is provided here for better understanding of the technical choices and conventions given below. Messages are *published* (AMQP terminology for sent) to *exchanges* and *consumed* (read) from *queues*. The communication patterns are implemented by *binding* queues to exchanges, possibly through other intermediary exchanges. Each binding also specifies appropriate message routing rules [6].

Each message in AMQP has a metadata field *routing key*, which is used by routing rules. The most flexible type of exchange is the *topic exchange,* where the routing key is supposed to consist of multiple words separated by dots and the routing filters are routing keys with wildcard operators * (matches any single word) and # (matches zero or more words).

When a message is published to an exchange, it is immediately delivered to all the queues that are directly or indirectly bound to it. If no such queues exist, the message is discarded. It is therefore vital that all queues and any intermediary exchanges are created before any messages are sent.

To illustrate this, imagine a system shown in Figure 2. The TSO (in the role of the balancing market operator) publishes the *market_cleared* message through RabbitMQ to an exchange named *tso.market*. A BSP *bsp1* might set up its own exchange *bsp1.market* and bind it to *tso.market* using appropriate routing rules so that only the messages that *bsp1* is interested in are routed to *bsp1.market*. Then, for each person from the team that should know about gate closure, *bsp1* might set up a queue *bsp1. personX*. When a person's working shift starts, her queue can be bound to the *bsp1.market* exchange and unbound when the shift ends, so that all the alarms will always be routed to all the currently present people.

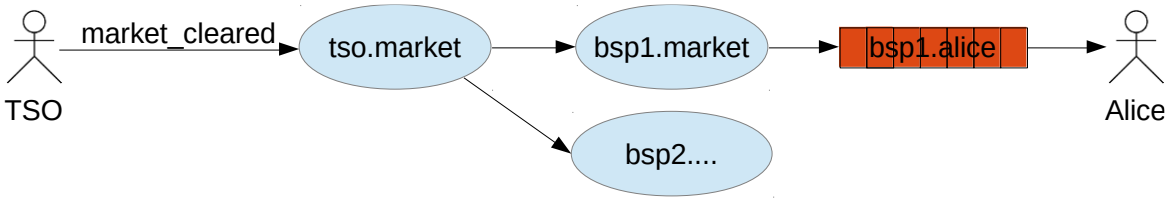

Figure 2: An example of message routing in RabbitMQ. Exchanges and queues are shown in blue and red, respectively.

Each communicating entity must have its/her own RabbitMQ account. Permissions are defined on individual exchange/queue names or regular expressions encompassing multiple names. To continue with the above example, the permissions should be:

user	read permissions	write permissions
TSO	*none*	tso.*
bsp1	tso.market;bsp1.*	bsp1.*
Alice	.*alice	*none*

Thus, the TSO can write to the exchange tso.market. Bsp1 can read from there and also both read and write every RabbitMQ resource whose name begins *with bsp1*, such that it can route TSO's messages to Alice through an intermediate exchange. Alice can only read messages intended for her and not write anything.

If we had multiple TSOs, either each of them must have an own exchange to publish to, or we have to trust them that one will not spoof the messages of the other.

5.2 Communication Patterns in eBADGE

In eBADGE we have currently identified the need for the one-to-one pattern (for example, the HEH sends a load report to its controlling entity) and one-to-many (as in the above example, the market operator signals gate closure to all BSPs).

In no case can we trust that the users will not spoof messages or eavesdrop on others. Therefore, for each possible pair <sender, receiver(s)[13]> we need a separate exchange which only this sender is allowed to write to and only this receiver(s) is (are) allowed to read.

Therefore, we need the following exchanges at the HEH level:

- for each HEH, a pair of exchanges for sending messages to/receiving from the HEH,
- for each entity using the services provided by the HEH operator (e.g., a VPP), a pair of exchanges for sending messages to/receiving from the entity.

5.3 Usage Conventions

5.3.1. Usernames and credentials

Each entity must have a unique name and a unique RabbitMQ username, which is recommended to be the same as the entity name. In the eBADGE pilot, for example, the HEH with MAC address B8:27:EB:EA:02:C1 has the RabbitMQ username *heh-b827ebea02c1* and the HEH operator Telekom Slovenije (TS) has the RabbitMQ username *ts.*

Each entity must have a client certificate and use it to authenticate with the RabbitMQ server. Password-based connections must be disabled.

The default account *guest* must be disabled.

5.3.2. Exchange naming and permissions

For each possible pair <sender, receiver> with a single receiving entity there should be an exchange *sender_name-to-receiver_name.* The sender and receiver should have write and read permission, respectively, and exactly one of them should have the configure permission. For example, a HEH B8:27:EB:EA:02:C1 that is operated by TS sends its measurements to the exchange *heh-b827ebea02c1-to-ts.* TS should have configure permission on this exchange.

In general, users can communicate with multiple other users, e.g., a HEH operator will control many HEHs. The described scheme is compatible with RabbitMQ's authorization system that uses regular expressions to define permitted resources for each user. Each user must have:

- read permissions on '.*-*to-username*' (any resource whose name ends with *'-to-username'*, where *username* is this user's RabbitMQ username),
- write permissions on 'username-to-.*'.

This simple scheme allows us to automate permission management for all users.

For each one-to-many communication pattern with multiple receivers, the group of receivers must also have a unique name and the exchange *sender_name-to-group_name* should be used. All members of the receivers must thus have read permissions for this exchange. However, this should only be used with caution. As will be explained in Section **Error! Reference source not found.**, RabbitMQ permission changes may not take effect immediately.

[13] Note that receiver(s) can also be a group, such as all BSPs that should be notified of a market clearing event.

hen an entity must not be allowed anymore to read messages intended for a certain group, it is thus not enough to revoke its permission but rather its RabbitMQ account must be blocked altogether. The use case for this is e. g. a BSP that has lost its license and must therefore immediately be banned from any participation in the market.

All exchanges and queues should be created as durable so that they survive a restart of the message bus server [7]. However, additional non-critical queues for messages processed elsewhere may be non-durable. For example, an entity might route all messages to the processing software as well as to a debugging tool that shows the live stream of messages. The latter may use a non-durable queue.

5.3.3. Provisions for customized message routing

The receiver can freely organize the further flow of messages intended for her using additional, private exchanges. For example, if a telecommunications provider owns and operates HEHs, they may first route messages from all the HEHs to a single exchange, then further route just the energy measurements to a database writer component, the error messages to an e-mail alert application, and all messages to a logging system. Therefore:

- a subset of exchange/queue namespace '*private-username-.**' is allocated to each entity to be used for such purposes, and the entity has full read, write, and configure permissions on this subset,
- no messages may be sent to the default (nameless) exchange, because the latter does not allow the sender to customize the routing in any way,
- all non-private exchanges must be of the type *topic*.

The first word of routing key of each message must be equal to the message type, i. e., the *msg* field of the eBADGE data model. In messages containing the *heh_id* field (e. g., a *load_report* message that contains measured values from a particular HEH and is sent by the HEH operator to a VPP), this field must be the second word of the routing key.

Figure 3 shows an example of message routing. Two HEHs are sending the messages to their respective outgoing exchanges. The entity ts, which controls the HEHs, routes all messages of type *report* (regardless of source HEH) into a metering database. All messages from *heh-42* (regardless of type) are routed to a debug console to facilitate a debugging session with this HEH.

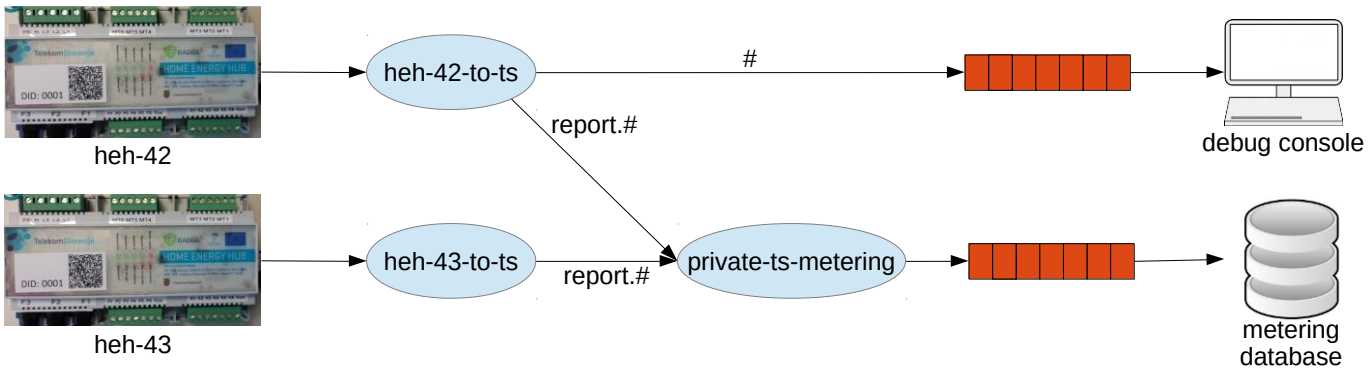

Figure 3: A possible message routing in eBADGE pilot.

6. The eBADGE Message Bus – Implementation, Final Version

Having selected RabbitMQ for the final version of the eBADGE MB, we have to:

- extend it with any unsupported but required functionalities,
- provide extensions to simplify its usage in the context of eBADGE.

The deployment and operation of the pilot has shown that RabbitMQ does not lack any of the required functionalities. Extensive non-functional testing of both RabbitMQ as well as its OpenADR counterparts XMPP and HTTPS was carried out in the project. As reported in [3], the RabbitMQ server software was not found to have any security, reliability, or performance flaws. However, to achieve the required properties, it has to be deployed and configured in an appropriate way. The below specifications and how they were derived during the preliminary testing provide all the necessary details.

6.1 Security

RabbitMQ's security mechanism has a documented disadvantage that any revoked permissions might not propagate to the already connected queues. That is, if an entity is permitted to read a certain message queue/exchange and subscribes to it, but the permission is later revoked, it is only guaranteed that the entity will not be allowed to re-connect; it may still be able to receive messages until disconnection, which in the worst case can be until restart of the RabbitMQ server. This is not unusual and, for example, Linux file permissions work in the same way.

The consequence for the eBADGE pilot is that the security mechanisms cannot be used to enforce access control to the services provided by the HEH operator to other entities. For example, when a home owner revokes a VPP the right to switch off her appliances, this must take effect immediately. This is the reason why each pair of communicating entities must have a separate exchange.

6.2 Reliability

The eBADGE message bus ensures the reliability of communications in the following ways:

1. It prescribes that a cluster of RabbitMQ servers is to be used, rather than a single server.
2. It prescribes that a high-availability server with automatic failover is to be used as a proxy to the actual cluster.
3. It implements a client-side reliability layer that handles the reconnects related to high availability as well as short-term client-side connection losses.
4. To avoid too many messages queueing in the server and compromising stability, it prescribes a limited message lifetime.

The details of these approaches are given below.

6.2.1. Clustering RabbitMQ Server

RabbitMQ clustering is based on Erlang's ability to distribute applications to multiple nodes. The basic set-up process is relatively straightforward [8] and results in a scalable cluster – its capacity can be increased or decreased simply by adding or removing nodes.

If a node in the cluster fails, the clients connected to it can simply re-connect to one of the remaining nodes. However, in a default configuration the associated message queue will be lost. The messages sent before the failure but not yet processed by the receiver, as well as those sent after the failure but before the re-connect, will be lost. To ensure a reliable operation, therefore, the following queue replication policy must be set:

Parameter	Value	Description
ha-mode	exactly	2 or 3 replicas (copies) of each queue (together with all messages) will exist. When a node containing a replica fails, another node is chosen (if available) for a new replica.
ha-params	2 or 3	
ha-sync-mode	automatic	When a new replica is created, it is also "synced", i.e. all messages in the queue

		are also copied. The queue will be unresponsive during the copying, but all replicas will be the same and thus no messages will be lost regardless of which node the client reconnects to.

In RabbitMQ versions older than 3.5, the "ha-mode exactly" parameter only ensured the initial number of replicas. New replicas were not automatically created on node failures and no appropriate mechanism existed for notifying the clients about node failures, thus handling creation of new replicas at the application level was also impossible. This was fixed in version 3.5, possibly influenced by the feedback from the eBADGE project[14].

6.2.2. Transparent Clustering

Rather than the client having to know about the cluster and deciding which cluster node to connect to, a high-availability proxy service should be used. The client will then see it as a single, highly available RabbitMQ server.

We recommend the open source tool haproxy. version 1.5.3. or newer[15]. It must be configured to know about all the nodes in the RabbitMQ cluster. It will then periodically check which of them are still alive and proxy the clients to all of the working nodes using a suitable load-balancing algorithm.

Although haproxy is simple, stable, well-tested and field-proven software, a server running it is still susceptible to failure for various reasons. Therefore, at least two haproxy nodes should be used. To maintain the transparency, a VRRP (Virtual Router Redundancy Protocol [9]) implementation must also be used. We recommend VRRPd version 1.0-2 or newer[16],

VRRP allows multiple servers (in our case, two or more haproxy instances) in the same network to obtain the same (virtual) IP address, but with different priorities. The clients must always connect to this virtual IP address. Thus, if the higher-priority haproxy instance is working, the traffic for this IP will be routed there; otherwise, the traffic will be routed to the auxiliary (lower priority) haproxy instance.

The full reliability setup is shown in Figure 4. The normal operation is depicted in the left part. VRRP routes all connections to the primary haproxy1, which then distributes them among the nodes of the RabbitMQ cluster. The right part shows the connection when both haproxy1 and one of the RabbitMQ nodes fail. VRRP routes all new connections to the remaining haproxy2, which then distributes them only among the working RabbitMQ nodes. At the time of failure all connections that use the failed node are dropped, which has to be dealt with at the client side, as is explained in the next subsection.

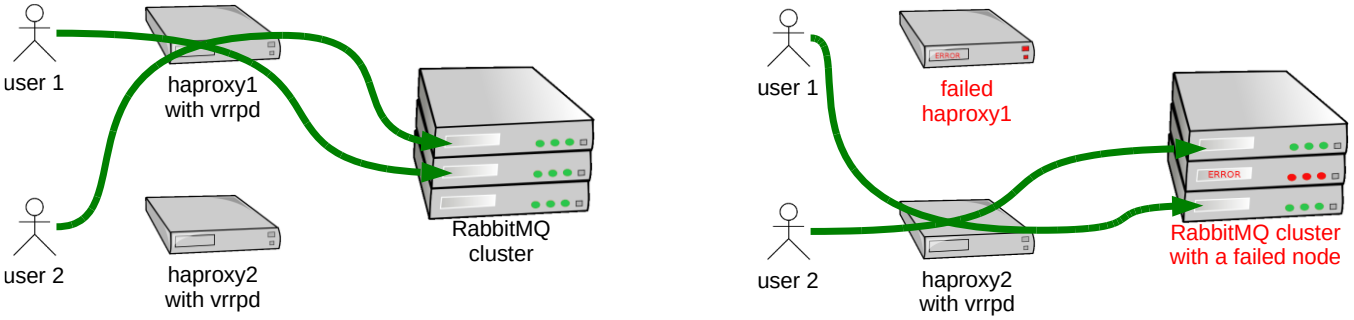

Figure 4: Transparent clustering in eBADGE message bus: normal operation (left) and partial system failure (right).

6.2.3. Cient-Side Reliability Layer

When a cluster node fails, all of the clients that were connected to it will drop the connection. The client-side library used in the pilot[17], as well as most probably other client-side libraries, leaves handling the connection loss to the application level.

[14] On January 8th 2015 we reported the shortcoming to the official mailing list rabbitmq-users@googlegroups.com.
[15] http://www.haproxy.org/
[16] http://www.fwbuilder.org/4.0/docs/users_guide5/vrrpd_cluster.shtml

The problems can be solved well on the client side, thus we have implemented a client-side reliability layer that periodically tries to re-connect when a connection is dropped and buffers the outgoing messages for a short period. This layer is part of the eBADGE message bus and is transparent to the HEH programmer. The HEH programmer therefore is not concerned at all with the reconnect process. The same layer also hides temporary (up to one minute) connectivity loss on the client side, therefore the HEH programmer is not concerned with those either.

6.2.4. Long-Term Client Outage

The in-field testing showed that the RabbitMQ server becomes unstable once the undelivered messages fill the available memory, for example, when the listener process in the VPP crashes but the HEHs continue to send measurements. This was remedied by setting the TTL (time-to-live) parameter for all messages on all queues. Deleting messages that could not be delivered within 15 minutes proved a reasonable choice, because it is enough time to buffer computer restarts, maintenance, etc, but short enough that the measurements cannot fill the RabbitMQ server's memory.

There are two potential significant problems with deleting messages:

- activation commands will be missed. However, a 15-minute-old activation command is already out-of-date and is safer ignored by the HEH than acted upon.
- measurements will be dropped. However, the HEH buffers the measurements in memory for multiple hours and the data model allows requesting re-sending of past measurements, thus this problem is solvable on the measurement collector's side.

6.3 Mapping Messages to Objects, and Vice Versa

To simplify the development of eBADGE software components, the message bus includes a reference implementation of the eBADGE data model [1]. This is implemented as a Python library that maps each message type specified in the eBADGE data model 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 model. For example, Python *datetime* objects typically do not have a specified time-zone, but the data model forbids this to avoid confusion. Also, certain field names in eBADGE messages are reserved names in Python, also requiring special processing.

6.4 Sending and Receiving eBADGE Messages

The eBADGE message bus, i.e. the Python library developed on top of RabbitMQ, 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 code excerpt shown in Figure 5 illustrates, usage is straightforward and our API is transparent in that it can be used in conjunction with existing RabbitMQ objects and features rather than hiding them.

[17] We use pika, a pure Python implementation of AMQP 0.9.1. See https://github.com/pika/pika for details.

```
import Queue
import threading
from ebadge_msg.comm import AbstractConsumer, Connector_Settings, Connector_async
from ebadge_msg.heh_level import Report

send_queue = Queue.Queue()
queue_lock = threading.Lock()

class Consumer(AbstractConsumer):
    def on_get_report(self, request, method=""):
        values = dict()
        for sig in request.signals:
            values[sig] = [0.12, 0.17, 0.33, 0]

        report = Report(
            request.from_, request.to,
            request.resolution, request.device,
            values)

        queue_lock.acquire()
        send_queue.put_nowait(report)
        queue_lock.release()

conn_settings = Connector_Settings("rabbitmq.server.address", "b827ebea02c1", location="heh")
send_thread = Connector_async.connector(conn_settings, workqueue=send_queue, queueLock=queue_lock)
send_thread.start()
recv_thread = Connector_async.connector(conn_settings, listener=Consumer())
recv_thread.start()
```

Figure 5: A simple program that uses the eBADGE message bus

This code first declares a queue for outgoing connections and a corresponding lock that must be used when adding messages to the queue. It then declares a Consumer class for eBADGE messages. As implemented here, whenever this consumer receives the *get_report* message, it responds with a *Report* object containing the requested signals in requested time period; however, the actual values are only hard-coded examples. The consumer then puts this object into the outgoing queue. The main part of the program creates two threads that will take care of actually sending and receiving the messages.

Behind the scenes our library will now wait for incoming eBADGE messages on. When a *get_ report* message arrives, the corresponding method of our *Consumer* will be called. Afterwards, the library will take care of encoding the new *Report object* into an eBADGE message type *report.* Furthermore, as described above, it will transparently provide the auto-reconnect feature and buffering of outgoing messages.

6.5 Obtaining the Software

The reference Python implementation of the eBADGE data model is available on GitHub:
https://github.com/eBADGE/message-bus
As explained in the included *README.md* file, it requires Python 2.7 and pika 0.9.8 (Python client for RabbitMQ, installable through the Python package installer *pip*). Short instructions on set-up and running are also included in the same file.

7. Conclusions

We have put forward the requirements for the eBADGE message bus, including the quantitative performance requirements that will allow the pilot project to demonstrate scalability to production-size deployments with tens of thousands of homes willing to participate in demand response aggregated into a VPP, and a large number of such VPP participating in the market. We decided to use an architecture with a single or multiple central message proxies/brokers to simplify configuration and management of the HEHs and other communicating end-points in the pilot.

A short list of considered open-source message bus technologies consists of RabbitMQ, XMPP, OpenAMQ, ActiveMQ, and Apollo. The performance comparisons available in the literature, limitations that some of the technologies suffered during the time of our tests and our analysis of network packet sizes suggest RabbitMQ, XMPP and ActiveMQ as suitable candidates. We selected RabbitMQ for the first implementation and further evaluation in the real life pilot environment, and kept it throughout the project, thus resulting in the final version of the eBADGE message bus

We have performed limited tests of RabbitMQ as-is, without any extensions, both in laboratory and with the HEHs deployed in the field. We confirmed that it satisfies all of our functional requirements as well as the non-functional requirements needed in the pilot itself. Related tests also done in the eBADGE project confirm that RabbitMQ also has certain advantages over XMPP.

The final version of the eBADGE message bus thus consists of a RabbitMQ cluster installation, high-availaiblity proxy server, a specification of the RabbitMQ resources that should be used for communication with appropriate naming conventions and access permissions, and a reference implementation of the eBADGE data standard. The latter is implemented as a Python library that provides a two-way mapping between eBADGE messages and Python objects and a simplified AMQP-like API for sending and receiving messages. The library also provides a transparent client-side reliability layer. We made the Python library publicly available under a free software license.

References

[1] eBadge Project, “The eBadge Data Model Report - Final Version (deliverable 3.1.3),” 2015.

[2] eBadge Project, “The eBadge Message Bus - Second Intermediate Version (deliverable 3.2.2),” 2014.

[3] eBadge Project, “The eBADGE ICT Interfaces Testing report - Final version,” 2015.

[4] A. Mihailescu, “Messaging Systems – How to make the right choice?,” [Online]. Available: http://rivierarb.fr/presentations/messaging-systems/. [Accessed 17 5 2013].

[5] M. Salvan, “A quick message queue benchmark,” [Online]. Available: http://x-aeon.com/wp/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/. [Accessed 17 5 2013].

[6] Pivotal Software, Inc., “RabbitMQ – AMQP 0-9-1 Model Explained: An Online Guide,” [Online]. Available: https://www.rabbitmq.com/tutorials/amqp-concepts.html. [Accessed 11 9 2014].

[7] Ruby RabbitMQ Client Maintaners Team, “Durability and related matters: An Online Guide,” [Online]. Available: http://rubybunny.info/articles/durability.html. [Accessed 16 9 2014].

[8] Pivotal Software, Inc., „RabbitMQ - Clustering Guide,“ [Elektronski]. Available: https://www.rabbitmq.com/clustering.html. [Poskus dostopa 04 03 2015].

[9] S. Knight, D. Weaver, D. Whipple, R. Hinden, D. Mitzel, P. Hunt, P. Higginson, M. Shand in A. Lindem, „Virtual Router Redundancy Protocol (RFC2338),“ The Internet Engineering Task Force, 1998.