The eBADGE Message Bus
First Intermediate Version
Deliverable report
## Document Information

<table>
<thead>
<tr>
<th>Programme</th>
<th>FP7 – Cooperation / ICT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Project acronym</strong></td>
<td>eBADGE</td>
</tr>
<tr>
<td><strong>Project Full Title</strong></td>
<td>Development of Novel ICT tools for integrated Balancing Market Enabling Aggregated Demand Response and Distributed Generation Capacity</td>
</tr>
<tr>
<td><strong>Grant agreement number</strong></td>
<td>318050</td>
</tr>
<tr>
<td><strong>Number of the Deliverable</strong></td>
<td>D3.2.1</td>
</tr>
<tr>
<td><strong>WP/Task related</strong></td>
<td>WP3 / T3.2</td>
</tr>
<tr>
<td><strong>Type (distribution level)</strong>&lt;sup&gt;1&lt;/sup&gt;</td>
<td>PU</td>
</tr>
<tr>
<td><strong>Due date of deliverable</strong></td>
<td>30.09.2013 (Month 12)</td>
</tr>
<tr>
<td><strong>Date of delivery</strong></td>
<td>14.1.2014</td>
</tr>
<tr>
<td><strong>Status and Version</strong></td>
<td>Final, v2.1</td>
</tr>
<tr>
<td><strong>Number of pages</strong></td>
<td>22 pages</td>
</tr>
<tr>
<td><strong>Document Responsible</strong></td>
<td>Marjan Šterk - XLAB</td>
</tr>
<tr>
<td><strong>Author(s)</strong></td>
<td>Daniel Vladušič-, Staš Strozak, Marjan Šterk - XLAB Radovan Sernec, Marko Rizvič - TS Ingo Sauer, Armin Fuchs, Andreas Flür - Sauper</td>
</tr>
<tr>
<td><strong>Reviewers</strong></td>
<td>Peter Nemček - CG</td>
</tr>
</tbody>
</table>

---

<sup>1</sup> PU: Public<br>RP: Restricted to other programme participants (including the Commission Services)<br>RE: Restricted to a group specified by the consortium (including the Commission Services)<br>CO: Confidential, only for members of the consortium (including the Commission Services)
### Issue record

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author(s)</th>
<th>Notes</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>03.01.2014</td>
<td>Marjan Šterk, XLAB</td>
<td>Glossary added, table of acronyms expanded, a few typos corrected</td>
<td>Final v.2.1.</td>
</tr>
<tr>
<td>2.0</td>
<td>13.12.2013</td>
<td>Renata Blatnik, XLAB</td>
<td>Project deliverable template applied</td>
<td>Final v.2.0</td>
</tr>
<tr>
<td>1.8</td>
<td>13.12.2013</td>
<td>Marjan Šterk, XLAB</td>
<td>Integrated comments by other partners</td>
<td>Draft</td>
</tr>
<tr>
<td>1.7</td>
<td>13.12.2013</td>
<td>Radovan Sernec, TS; Peter Nemček, CG</td>
<td>Reviewed, multiple improvements throughout</td>
<td>Draft</td>
</tr>
<tr>
<td>1.6</td>
<td>12.12.2013</td>
<td>Marjan Šterk, XLAB</td>
<td>Re-written most of the document according to recommendations from project review; title changed</td>
<td>Draft</td>
</tr>
<tr>
<td>1.5</td>
<td>12.12.2013</td>
<td>Ingo Sauer, Sauper</td>
<td>Added message overhead test report</td>
<td>Draft</td>
</tr>
<tr>
<td>1.1</td>
<td>16.10.2013</td>
<td>Marjan Šterk, XLAB</td>
<td>Format partially unified with other deliverables</td>
<td>Final v.1.1</td>
</tr>
<tr>
<td>1.0</td>
<td>30.09.2013</td>
<td>Marjan Šterk, XLAB</td>
<td>Summary and conclusions added, text polished for submission</td>
<td>Draft</td>
</tr>
<tr>
<td>0.3</td>
<td>25.09.2013</td>
<td>Marjan Šterk, XLAB</td>
<td>Most parts written</td>
<td>Draft</td>
</tr>
<tr>
<td>0.2</td>
<td>05.09.2013</td>
<td>Marjan Šterk, XLAB</td>
<td>Document outline created</td>
<td>Draft</td>
</tr>
<tr>
<td>0.1</td>
<td>12.06.2013</td>
<td>Daniel Vladašič, XLAB</td>
<td>Initial message bus comparison report</td>
<td>Draft</td>
</tr>
</tbody>
</table>

**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.

The content of this document reflects only the authors’ views and the European Commission is not liable for any use that may be made of the information contained herein.

The contents of this document are the copyright of the eBADGE consortium.
Table of Contents

Document Information ............................................................................................................................................. 2
Table of Contents .................................................................................................................................................... 4
Table of Figures .......................................................................................................................................................... 5
Table of Tables .......................................................................................................................................................... 6
Table of acronyms ..................................................................................................................................................... 7
Glossary ..................................................................................................................................................................... 8
eBADGE Project ...................................................................................................................................................... 9
Executive Summary .............................................................................................................................................. 10

1. Introduction ..................................................................................................................................................... 11
   1.1 Approach ................................................................................................................................................... 11
   1.2 Relation to the Rest of the Project ............................................................................................................. 11
   1.3 Outline of This Report .............................................................................................................................. 11
2. Requirements ................................................................................................................................................. 12
   2.1 Performance Requirements ....................................................................................................................... 13
3. Communication Architecture ............................................................................................................................ 14
   3.1 Choice of a “Proxy” System ....................................................................................................................... 14
4. Selection of Message Bus ............................................................................................................................... 16
   4.1 Performance Comparison ........................................................................................................................... 16
      4.1.1. Evaluation of RabbitMQ advanced message queuing protocol (AMQP) communication .................. 17
      4.1.2. Evaluation of ActiveMQ OpenWire communication ........................................................................ 17
   4.2 Current Choice ........................................................................................................................................... 17
5. The eBADGE Message Bus – Implementation, First Version ............................................................................. 18
   5.1 Preliminary Testing Results ......................................................................................................................... 18
   5.2 Mapping Messages to Objects, and Vice Versa ......................................................................................... 18
   5.3 Sending and Receiving eBADGE Messages ............................................................................................... 18
   5.4 Obtaining the Software .............................................................................................................................. 19
6. Conclusions ....................................................................................................................................................... 20
   6.1 Planned Updates .......................................................................................................................................... 20
7. References .......................................................................................................................................................... 21
Table of Figures

Figure 1: A typical ESB stack .................................................................................................................15
Figure 2: A simple program that uses the eBADGE message bus ..........................................................19
Table of Tables

Table 1: A list of the considered MQs........................................................................................................16
# Table of acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACER</td>
<td>Agency for the Cooperation of Energy Regulators</td>
</tr>
<tr>
<td>ACK</td>
<td>Acknowledgement</td>
</tr>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BSP</td>
<td>Balancing Service Provider</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DSO</td>
<td>Distribution System Operator</td>
</tr>
<tr>
<td>ESB</td>
<td>Enterprise Service Bus</td>
</tr>
<tr>
<td>HEH</td>
<td>Home Energy Hub, a metering and controlling device developed in eBADGE WP4</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>ICT</td>
<td>Information and Communication Technologies</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td>LTE</td>
<td>Long Term Evolution (4G mobile technology)</td>
</tr>
<tr>
<td>MB</td>
<td>Message Bus</td>
</tr>
<tr>
<td>MQ</td>
<td>Message Queue</td>
</tr>
<tr>
<td>MQTT</td>
<td>MQ (Message Queuing) Telemetry Transport</td>
</tr>
<tr>
<td>PKI</td>
<td>Public-Key Infrastructure</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>STOMP</td>
<td>Simple (or Streaming) Text Oriented Messaging Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transfer Control Protocol</td>
</tr>
<tr>
<td>TSO</td>
<td>Transmission System Operator</td>
</tr>
<tr>
<td>VPP</td>
<td>Virtual Power Plant</td>
</tr>
<tr>
<td>WP</td>
<td>Work Package</td>
</tr>
<tr>
<td>WS</td>
<td>Web Service</td>
</tr>
<tr>
<td>XMPP</td>
<td>Extensible Messaging and Presence Protocol</td>
</tr>
</tbody>
</table>
## Glossary

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>.NET</td>
<td>a software framework and run-time environment for multiple programming languages (e.g. C#, Visual Basic.NET), developed by Microsoft</td>
</tr>
<tr>
<td>C</td>
<td>a low-level programming language</td>
</tr>
<tr>
<td>C++</td>
<td>an object-oriented programming language based on C</td>
</tr>
<tr>
<td>data model</td>
<td>a specification of data types (objects) and the fields they contain; in this document refers to the eBADGE data model [eBADGE D3.1.1, 2013]</td>
</tr>
<tr>
<td>end-to-end encryption</td>
<td>encryption of messages at the sender side and decryption on the receiver side, such that no intermediaries can see unencrypted contents</td>
</tr>
<tr>
<td>enterprise service bus</td>
<td>a software architecture and corresponding multi-layer software stack for communication between mutually interacting software applications in a service-oriented architecture</td>
</tr>
<tr>
<td>failover</td>
<td>the act where running instances of a replicated entity take over from failed instance</td>
</tr>
<tr>
<td>fibre-to-the-home</td>
<td>optical fibre that runs directly to a home or office, typically used for Internet access</td>
</tr>
<tr>
<td>firewall</td>
<td>a (hardware or software) network device that blocks disallowed communication, typically based on IP addresses and port numbers</td>
</tr>
<tr>
<td>heartbeat</td>
<td>messages that are periodically sent in order to let the receiver know that the sender is still there</td>
</tr>
<tr>
<td>Java</td>
<td>a programming language and corresponding run-time environment developed by Oracle (originally by Sun Microsystems)</td>
</tr>
<tr>
<td>market model</td>
<td>a mathematical model of balancing energy market, being developed in eBADGE WP2</td>
</tr>
<tr>
<td>market simulator</td>
<td>a software simulator of the market model, being developed in eBADGE WP2</td>
</tr>
<tr>
<td>message bus</td>
<td>a generic name for a messaging proxy, the required server and client software, typically also capable of one-to-many communication</td>
</tr>
<tr>
<td>message header</td>
<td>the part of a message that contains the message metadata and typically precedes the payload</td>
</tr>
<tr>
<td>message payload</td>
<td>the part of a message that contains the actual data and typically follows the header</td>
</tr>
<tr>
<td>message queue</td>
<td>a data structure within a messaging proxy where messages are buffered until they are routed to the destination end-point</td>
</tr>
<tr>
<td>message server</td>
<td>an intermediary server that routes messages from a sending entity (end-point) to the receiving entity in the absence of direct connection</td>
</tr>
<tr>
<td>messaging proxy</td>
<td>an intermediary entity (typically, a server) through which one entity (end-point) can communicate with another without requiring a direct connection</td>
</tr>
<tr>
<td>Perl</td>
<td>a family of high-level programming languages</td>
</tr>
<tr>
<td>proxy</td>
<td>an intermediary entity through which one entity (end-point) can reach another entity in the absence of direct access</td>
</tr>
<tr>
<td>Python</td>
<td>a high-level programming language</td>
</tr>
<tr>
<td>Raspberry Pi</td>
<td>the type of single-board computer used in the eBADGE HEH</td>
</tr>
<tr>
<td>replication</td>
<td>use of multiple instances of the same entity so that if one instance fails the rest can take over</td>
</tr>
<tr>
<td>round-trip time</td>
<td>the time between sending a message and receiving a response</td>
</tr>
<tr>
<td>Ruby</td>
<td>a high-level programming language</td>
</tr>
<tr>
<td>serialization/deserialization</td>
<td>the process of transforming a software object to/from a stream of bytes</td>
</tr>
<tr>
<td>service delivery broker</td>
<td>a control layer or service that mediates access to services and resources</td>
</tr>
<tr>
<td>svn</td>
<td>a software versioning and revision control system by the Apache Software Foundation, also known as Subversion</td>
</tr>
<tr>
<td>ZeroMQ</td>
<td>a high-performance asynchronous messaging library</td>
</tr>
</tbody>
</table>
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
Sauper Umweltdatentechnik 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 for communication among all the entities in the eBADGE pilot project and potentially applicable to smart grid marketplace in general. The first version consists of RabbitMQ, an open source AMQP 0.9.1 implementation, as the messaging proxy 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 first version of the eBADGE message bus. We list the requirements, including a quantitative minimum performance specification required to show the scalability of the approach to production-sized deployments. 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 additional network overhead analysis done within our study. After a careful evaluation we have selected RabbitMQ as one that fits well within the eBADGE project objectives and constraints.
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 messages, as defined in [eBADGE D3.1.1, 2013].

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

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 are analysed and a market simulator is being 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 analyzing 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 standard.

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. The current version already achieves certain aspects of this goal.

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 will be 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.

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 4 compares selected four message bus implementations. Section 5 describes the actual implementation of the first version of the eBADGE message bus, that is, RabbitMQ with a Python library on top of it. It also provides the address where this library is available.
2. Requirements

As is described in [eBADGE D3.1.1], the communication in the eBADGE pilot can be divided into two isolated layers:

- field-level or home energy hub (HEH) – level, where the HEHs communicate with their controlling VPP and, optionally, with DSOs and other distribution-level entities,
- market-level, where the balancing service providers (BSPs) communicate with the TSOs.

The HEH-level is both more demanding 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 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 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 one of the communicating parties (in our case by VPP) 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.
4. The given approximate message sizes were estimated based on the eBADGE data model [eBADGE D3.1.1, 2013]. 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 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.

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 spite of the above, we also considered ZeroMQ because it is generally seen as extremely high-performance. However, only the outdated version 2.1 is currently 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.

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/](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](http://zeromq.org/bindings:clr)
Figure 1: A typical ESB stack\(^4\)

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 AMQP 0.9.1 and 1.0. Please note these 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Evaluated Version</th>
<th>Language Bindings</th>
<th>Supported Standards / Protocols</th>
<th>Native</th>
<th>Others&lt;sup&gt;5&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>RabbitMQ&lt;sup&gt;6&lt;/sup&gt;</td>
<td>3.1.3</td>
<td>.NET, Java, C/C++, Python, Perl, and many more</td>
<td>AMQP 0.9.1</td>
<td>STOMP, MQTT, XMPP, REST</td>
<td></td>
</tr>
<tr>
<td>ActiveMQ&lt;sup&gt;7&lt;/sup&gt;</td>
<td>5.8</td>
<td>Java, C/C++, .NET, Python, and many more</td>
<td>OpenWire</td>
<td>AMQP 1.0, REST, STOMP, WS Notification, XMPP, MQTT</td>
<td></td>
</tr>
<tr>
<td>OpenAMQ&lt;sup&gt;8&lt;/sup&gt;</td>
<td>1.4c1</td>
<td>C, Python&lt;sup&gt;9&lt;/sup&gt;</td>
<td>AMQP 0.9.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apollo&lt;sup&gt;10&lt;/sup&gt;</td>
<td>1.6</td>
<td>Java, C/C++, Python, .NET, and more</td>
<td>AMQP 1.0</td>
<td>REST, STOMP, WS Notification, XMPP, OpenWire, MQTT</td>
<td></td>
</tr>
</tbody>
</table>

Table 1: A list of the considered MQs.

OpenAMQ was discarded due to lack of support for .NET and other languages. 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 [Mihailescu, 2013; Salvan, 2013]. These comparisons were done using the AMQP and STOMP protocols.

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<sup>11</sup>, as it records communication and also provides various options for analysis.

---

<sup>5</sup> Installation of additional plug-ins may be required.


<sup>7</sup> [https://activemq.apache.org/](https://activemq.apache.org/)

<sup>8</sup> [http://www.openamq.org/](http://www.openamq.org/)

<sup>9</sup> The web site also lists Java and Ruby but the links are dead or outdated.

<sup>10</sup> [http://activemq.apache.org/apollo/](http://activemq.apache.org/apollo/)

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.

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.

4.2 Current Choice

After working on the prototype for some time until we achieved almost the same application-level functionality 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.
5. The eBADGE Message Bus – Implementation, First Version

Having selected RabbitMQ for the first 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.

5.1 Preliminary Testing Results

Preliminary testing has not revealed stability, performance, or security problems. More extensive tests of stability, performance, and security will be performed in the next year in the Task 3.3 of the project.

RabbitMQ server replication was already tested and worked as advertised: a “cluster” of multiple RabbitMQ servers was deployed and when one of them failed the others took over its part of the load. However, since failover is not transparent to the client, we plan to implement a client-side reliability layer in the next version.

The only missing feature found so far is end-to-end encryption, which is optional for the eBADGE pilot. RabbitMQ does support 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. 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 on top of RabbitMQ. Whether we will implement it depends on further input from WP4 and 5.

5.2 Mapping Messages to Objects, and Vice Versa

To simplify the development of eBADGE software components, this first version of the message bus already includes a reference implementation of the eBADGE data model [eBADGE D3.1.1, 2013]. 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.

5.3 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 2 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.
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())

Figure 2: A simple program that uses the eBADGE message bus

This code first creates two eBADGE MB connectors, one for upward and one for downward communication. It then defines an implementation of a message consumer (based on the class AbstractConsumer) and starts it on the downward connector.

Behind the scenes our library will now wait for incoming eBADGE messages on downConnector. When a get_load_report message arrives, the corresponding method of our Consumer will be called, which will create a LoadReport object with hard-coded values and send it over the upConnector. The library will take care of encoding the object into an eBADGE message type load_report.

5.4 Obtaining the Software
The reference Python implementation of the eBADGE data model can be downloaded from https://dev.xlab.si/ebadge/ebadge-svn/wp3/message-bus/releases/1.0/ 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. It requires Python 2.7 and pika 0.9.8 (Python client for RabbitMQ, installable through the Python package installer pip). On the server side we currently use the default installation of RabbitMQ.
6. 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, OpenAMQ, ActiveMQ, and Apollo. The performance comparisons available in the literature, limitations of some of the technologies and our analysis of network packet sizes suggest both RabbitMQ and ActiveMQ as suitable candidates. We selected RabbitMQ for the first implementation and further evaluation in the real life pilot environment.

We have performed the first series of tests on RabbitMQ as-is, without any extensions. The missing feature is end-to-end encryption (optional for basic eBADGE implementation), which we will consider in the next two versions.

The first version of the eBADGE message bus thus consists of a default RabbitMQ installation 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. We made the Python library publicly available under a free software license.

6.1 Planned Updates

Please note that this is the first, prototype version. The next two versions are planned as yearly revisions. The eBADGE data model [eBADGE D3.1.1] can and will almost certainly be upgraded/extended and the message bus will be updated accordingly.

Other planned updates will include a client-side transparent reliability layer and support for .NET and Java.
References

