Implementation of a Scalable and Robust Messaging Solution for Flexibility Trading

Ferdinand von Tüllenburg, Jia Lei Du, Georg Panholzer
Salzburg Research Forschungsgesellschaft mbH
Salzburg, Austria
{ferdinand.tuellenburg, jia.du, georg.panholzer}@salzburgresearch.at

Rafael Vidal
Technological Institute of Aeronautics
So Jos dos Campos, Brazil
rafael.vidal125@gmail.com

***Abstract*—As Smart Grids highly depend on the information exchange between its components, messaging usually plays an important role. The messaging solutions used in Smart Grids are demanded to be scalable and robust. Scalable due to the usually large spatial extends and many communicating components, Robust due to the criticality of reliable information transfer for Smart Grid operation.**

In context of a flexibility coordination paradigm, where controllable and decentralized power sources and loads are utilized for stable grid operation, this article focuses on the development and implementation of a suitable message-oriented middleware solution. Though, particularly designed towards the system architecture and functional requirements or the flexibility coordination use case, the envisaged messaging solution aims at being applicable to a broad variety of Smart Grid applications. Apart from describing the components of the messaging system, also an application programming interface for the messaging solution is introduced. This interface is provided to component implementors with special attention to ease-of-use. Finally, we provide some performance considerations and evaluations on the developed system.

***Index Terms*—Flexibility Trading, Messaging Middleware, Smart Grid, Reliable Messaging**

I. Introduction

In future electrical power systems decentralized controllable power sources and loads will be coordinated in order to equalise production and consumption of electrical power and also optimise power flows through the grid. One control mechanism in this context is flexibility coordination using the capabilities of assets (e.g. photovoltaics, batteries, industrial loads) to time-shift consumption or production and/or adapt the amount of produced or consumed power.

In the ERA-Net Smart Grids Plus project Callia the concept of flexibility coordination is examined in an pan-European context. System operators at distribution (DSO) and transmission (TSO) level as well as owners of flexibility sources are linked together in a technical (through an ICT-system) and also a legal/business sense. Aim is to allow coordination of flexibilities within scope of a single distribution system (intra-DSO) or between multiple distribution systems (inter-DSO). The project investigates opportunities and challenges of collaborative grid balancing and develops suitable business models including flexibility market design and tradable products. At the end of the project the developed solutions will be implemented in a pan-European testbed including real controllable loads and distributed power sources.

As flexibility coordination highly depends on communication between grid operators, flexibility owners and their assets, one important research target of the Callia project is the development of a suitable messaging solution. Like many Smart Grid applications, the flexibility coordination use case also includes large spatial extends, comprising of large amounts of interconnected systems and devices, additionally to being a critical infrastructure. Due to this in particular scalability and robustness of the messaging systems is demanded.

From the perspective of a messaging system robustness means that information distribution is possible at least in non-affected parts of the system even in case of active failures within other parts of the messaging system. Scalability means to support large numbers of components (hardware and software) which need to exchange messages. This requires the messaging system to provide high throughput in terms of messages per time unit.

Due to aspects such as event-based character of information interchange, better separation of concerns during application development and its general applicability for different communication scenarios we argue for message-oriented middle aware (MOM) to be used in Smart Grids. This article focuses on the implementation of a message-oriented middleware suitable to but not limited to the flexibility coordination use case.

II. Related Work

In the past, message-oriented middleware has been proposed to be used for Smart Grid applications mainly due to the event-based character of information exchange. Smart Grid applications are particularly well supported by systems that use message-queuing to achieve asynchronous messaging and message prioritization [1]. Several competing solutions have been proposed, which can be generally divided in broker-based (such as Kafka [2], Active MQ [3] and [4]) and brokerless approaches such as Java Messaging Service (JMS) [5], NSQ [6] or ZeroMQ [7]). Broker-based systems use a central system as information hub between communicating peers, while brokerless systems mainly provide an API abstracting the details of network communication. For the sake of simpler connection management and system configuration we argue for using a broker-based system [8].

Apart from the stated MOM solutions, also cloud-based solutions such as Amazon SQS+SNS or solutions belonging to Microsoft Azure are available. However, it was decided to exclude those from our considerations, as power system operators traditionally provide their own communication services because they represent critical infrastructure.

Furthermore, broker-based message-queuing systems exist basically in two flavours: message-oriented and data-centric. Message-oriented solutions uses message queues to distributed messages from one or more senders to one or more interested receivers. Data-centric message queuing systems use a system data model shared between all parties and as the data is changed by one party this particular change is published to all interested parties. It has been recommended to apply a data-centric messaging solution (namely OMG Data Distribution Service (DDS) [9]) mainly due to its focus on in-time information distribution and built-in rich end-to-end quality of service (QoS)[10]. Particularly for Smart Grid applications, however, there are a few reasons why message-oriented approaches would be more suitable. First, even if means for end-to-end QoS are defined with DDS, those are hardly helpful in wide-area-network (WAN) environments due to limited knowledge about the network. Second, the DDS approach leads to a very strong coupling between messaging and application semantics leading to complex system descriptions with limited re-usability and extensibility. Especially due to the latter, following the vision of providing a generally applicable messaging solution, message-oriented systems seem to be the better starting point.

Focusing on broker-based message-oriented middleware solutions several products exist. For the usage particularly in Smart Grid scenarios only Harmony has support for quality of service (QoS) awareness for messaging in large-scale cyber-physical-systems (CPS) such as Smart Grids. [11]. The system consists of several interconnected broker instances distributed over a large geographical area where each broker instance provides a regional (local) connection point for field devices such as grid sensors. By employing network monitoring and overlay routing, the system optimizes the inter-broker communication regarding network resource usage and message delays respective to application requirements by supporting prioritisation. Although the stated features are attractive, the project seems not be supported anymore which is why it has been ruled out for our usage.

While the Harmony broker-based MOM is particularly designed for usage in Smart Grids field, several other broker-based MOMs have been developed in industry and research. Underneath those, particularly Apache Kafka [2] as well as ActiveMQ [3] and RabbitMQ [4] (both based on AMQP [12]), gained most attention in last years. Though not especially mentioned in context of Smart Grids, these solutions are frequently used in big data and Internet-of-Things applications making them interesting candidates for being used. However, due to their similarities a more detailed comparison is necessary (following in section IV.).

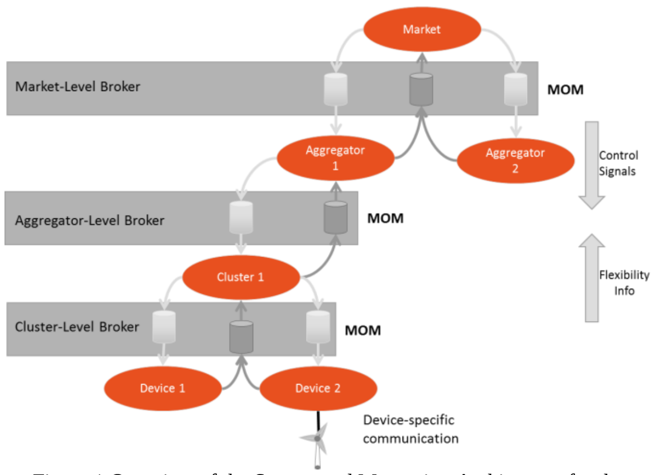

Figure 1 Overview of the System and Messaging Architecture for the flexibility-coordination use case.

III. SYSTEM AND MESSAGING ARCHITECTURE FOR FLEXIBILITY TRADING

Figure 1 shows the layered agent-architecture developed in the project. At the bottom layer (Flexibility Asset Layer) reside device agents providing a common control interface of power grid hardware (e.g. a photovoltaic system or a battery) to the flexibility coordination system. Additionally, device agents implement a device-specific interface in order to read status information (such as state of charge of a battery) directly from the device or send control commands (e.g. charge or discharge) towards the device.

Within the hierarchy, each device agent may be directly connected to exactly one agent at the cluster layer or aggregation layer. Both are serving the purpose of condensing and aggregating flexibility potentials of assets. The difference between both layers is that the cluster agents groups assets belonging logically together (e.g. situated in one facility) while aggregator agents groups whole clusters or individual assets due to contractual (business) relation. Aggregator agents trade flexibilities at the market on behalf of asset owners.

At the market platform the aggregator bids are processed and flexibility potential is exchanged with respect to an optimal grid operation. A certain set of flexibilities is selected for activation at the market platform from the transmitted bids with respect to the current grid state.

From the perspective of information flow, from device agents up to the market (upstream) flexibility information in form of device states or flexibility bids is transmitted. In the opposite direction (downstream), control signals for flexibility scheduling and activation are transmitted.

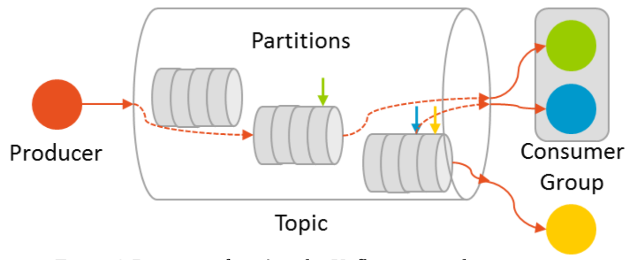

Figure 2 Diagram of an Apache Kafka topic and its partitions

The deployment of message-oriented middleware (MOM) instances within the system architecture is done as shown in Figure 1. Instances are deployed at each border between two vertically neighbored layers for each combined group. Thus, the MOM instances are provided and individually configured by either a cluster owner (e. g. a facility operator), an aggregator participating in flexibility trading or the market platform operator. This approach takes into account the security, robustness, scalability and also control of system. The MOM providers at each level have better knowledge about connected agents, for instance by exactly knowing device types (e.g. for cluster owners) or contractual relations (at market or aggregator level). Additionally, the connected agents at each MOM instance are limited to a small number compared to a centralized solution. Both aspects are especially important for large scale deployments where thousands of agents participate in flexibility trading.

IV. Message Brokering For Flexibility Trading

In order to find a good basis for our developments we compared the features of Apache Kafka, RabbitMQ and ActiveMQ. According to a study that has shown that only marginal differences (feature-wise and performance-wise) exist between ActiveMQ and RabbitMQ[13], both are considered jointly in the rest of this article.

The comparison has been done with regard to robustness, reliable message transfers, scalability and security. However, from a coarse-grained view onto these features, all are supported in slightly different ways by the considered MOM implementations. For instance in terms of security, all of them provide TLS encrypted communication between broker and senders or receivers. In terms of reliable message transfers, guaranteed messaging delivery is supported by both as well as message priorities (not directly by Kafka, but easily achievable by using different topics/partitions) and in-order-delivery. In terms of robustness, both AMQP-based brokers and Kafka can be operated in clusters as well as provide message persistence (using extension software for RabbitMQ and ActiveMQ). Clustering also is also a means for achieving scalability.

Going into more detail, however, reveals that Kafka has aspects more supportive of scalability and performance compared to AMQP-based approaches. The first aspect is the concept of partitions Kafka provides (see Figure 2). Partitions can be seen as message sub-queues within a single topic, where published messages get stored in an immutable first-in- first-out order. This allows for parallel writes into one topic particularly improving scalability with respect to multiple producers (senders) of messages. Published messages get sorted into exactly one certain partition of the topic - either directly specified by the producer or in a round-robin manner controlled by Kafka. For message consumption a consumer polls a topic and retrieves the messages of any of the partitions. At the receiver side, Kafka provides - similar to AMQP - consumer groups, speeding up processing of messages in a topic. Parallel reads and writes minimize the additional latency induced by message brokering in general.

Furthermore, Kafka provides high performance message persistence at scale. Kafka is using linear disk I/O provided by operating system cache paging. In doing so, Kafka persists every single message to disk before delivery ensuring durable messaging including message replay. Replaying messaged can be used by receivers which need to restart (e.g. due to failure) or even for traceability of the whole system.

Though an experiment revealed that messaging latency (time between producing and consuming a message) with Kafka is high compared to AMQP-messaging[1], it gets apparent that Kafka shows its strengths when large amounts of messages are sent due to batch-processing at sender and consumer side. Unfortunately, especially batching (sending / receiving of multiple messages at a time) is something which is considered as uncommon in Smart Grid applications. These operate more event-based (single messages are sent if a certain event happens).

However, as those delay-values for the single message test in the evaluation stated above appeared implausibly high, we decided to examine own tests with disabled batching (discussed in section VI). Also another evaluation (showing higher throughput and lower latency of Kafka compared to others) [14] raises doubts on those stated results. With respect to scalability, CPU load at the broker both stated experiments revealed higher frugality of Kafka even if a high load of messages are sent.

All in all, especially some technical details of Kafka's implementation as well as its' experimentally shown higher performance and scalability let us decided to bet on Kafka.

V. Implementation of Flexibility-Trading Messaging

Beside the specific Kafka broker deployment architecture and configuration for the flexibility trading application, a messaging middleware client API for inter-agent information exchange was developed. Thus, a very easy-to-use API is provided through which most of the advantages and features of the messaging solution can be used transparently. Additionally, an efficient message serialization system is included in the developed solution. The main design goals of the messaging client API was the provisioning of a high-level abstraction of messaging as well as message encoding serialization which allows agent developers to focus on

[1] https://dzone.com/articles/message-brokers-in-indirect-communication-paradigm

developing agent logic without the need to know about messaging details.

Before an agent is able to send or receive messages it must be registered at the Kafka broker including exchanges of necessary security credentials. Within the agent hierarchy, each agent has connections to at least one and up to two Kafka brokers - one broker for upstream data transfers and/or one for downstream. For this a minimal set of information needs to be provided in advance in form of a simple configuration file which is given in listing 1.

Listing 1 Configuration file of a messaging client

```
own.id=cluster-1
agent.upstream.id=aggregator-1
kafka.upstream=192.168.100.135:9093
security.upstream=TLS
kafka.downstream=192.168.100.136:9093
security.downstream=TLS
```

Beside an unique identifier (at broker-level), addressing information of the corresponding upstream and downstream Kafka instances, the ID of the upstream-level agent and the security configuration needs to be specified.

After the agent has successfully connected to a Kafka broker, the agent is ready to send or receive messages. The API provides simple methods for sending messages to and receiving messages from other agents. The agent developer need not be aware of the underlying persistency, guaranteed delivery, fault tolerance, and other mechanisms. The code examples in listing 2 for a sender and listing 3 for a receiver use the Python API. A functionally identical Java API is also available.

When sending a message, the API user only needs to provide the direction of sending or receiving (upstream, down- stream). The resolution of actual topic names is hidden by the API, which is able to automatically derive topic names by using the information given in the configuration file and the following common topic naming scheme:

1) The common topic for upstream messages is named using the ID of the upper-layer agent with suffix ”-in” appended. E.g. for agent ”cluster-1” the topic is named ”cluster-1-in”.
2) The names of private topics for downstream messages is created by appending the lower-layer agent ID to the own ID separated by a hyphen. E.g. the private topic for agent ”aggregator-1” and agent ”cluster-1” is named ”aggregator-1-cluster-1”.

For private topics the potential problem occurs that an upper- layer agent must know the names of any lower-layer agent prior to send messages. In case of our flexibility-trading application, however, this shortcoming is alleviated as an upper-layer agent is never required to initially contact a lower- layer agent. This situation is now exploited by inserting the ID of the sender agent into each application-level message. This allows an upper-layer agent to derive the correct private topic name.

Listing 2 Code for sending a message

```
from calliaMessaging import CalliaProducer
from calliaMessaging.examples import
        exampleMessages

calliaProducer=CalliaProducer.createFromConfig
        (configFile)
calliaProducer.send(exampleMessages
        [’FlexOffer’], UP)
```

Listing 3 Code for receiving a message

```
from calliaMessaging import CalliaConsumer

calliaConsumer=CalliaConsumer
        .createFromConfig(configFile)
msg = calliaConsumer.receive(UP)

print(”Received message ” + str(msg[”object”])+
” of type ” + msg[”type”] +
” from topic ” + msg[”topic”] +
” and sender ” + msg[”senderId”] +
” at ” + msg[”timestamp”])
```

With respect to scalability and efficiency of data transfers, Apache Avro [15] has been used for serializing and de-serializing application level messages into highly compact binary representations. The basic concept of Avro is using schema files (in JSON format) defining how application data is to be serialized into binary data, and how binary data is to be interpreted by a reader. In order to achieve efficient data transfers, the schema files are not part of the actual binary data but exchanged beforehand.

Generally, the developed messaging solution can be configured to a large extent by simply editing the configuration and/or schema files. As an example the communication topology can be changed by editing the broker IP addresses/agent IDs. Encryption can be switched on and off. And the content/- format of the exchanged messages can be changed by editing the Avro schema files. Thus application developer using the developed messaging system can focus on the business logic and mostly just configure the messaging system.

VI. Performance Evaluation

To get a clearer sight on the scalability and performance of Kafka with regard to the flexibility trading architecture and the event-based messaging scenario first tests have been performed. Especially the latter is an important difference to already published evaluations as Kafka gets much of its performance-boost from batch processing, which is not applicable in event-based messaging scenarios.

In the first test we were interested in the basic overhead regarding messaging latency the Kafka broker produces. This test gives us an impression about suitability of the solution for interactive and latency-sensitive applications. In the second test, a rough evaluation of the amount of messages which can be sent per second without batching has been carried out. This evaluates the scalability of our solution with respect to an event-based communication scenario with a multitude of independent communicating components.

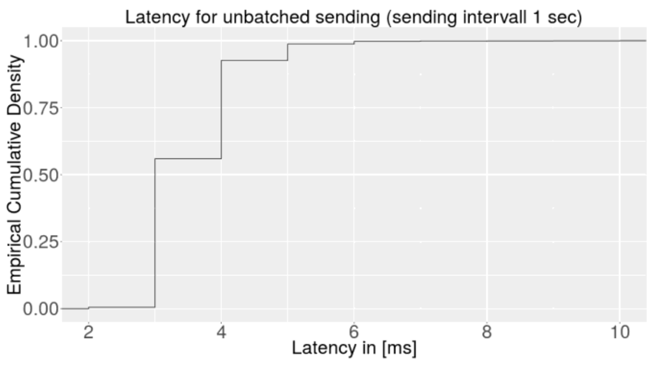

Figure 3 Empirical Cumulated Density Function of measured latency. Shows more than 95% of the latency values are blow 6 ms. Max. 10 ms.

TABLE 1 HOST CONFIGURATIONS

Name	CPU	# Cores	RAM
Producer	Intel Pentium G3250 @ 3,20 GHz	2	12 GB
Broker	Intel Core i5-4460S @ 2,90 GHz	4	16 GB
Consumer	Intel Pentium G3250 @ 3,20 GHz	2	8 GB

The test setup for both tests comprises one message producer instance, one single-instance Kafka messaging broker and one message consumer instance distributed over three physical machines. The hardware-specification is shown in table I. The broker system is equipped with a solid state drive (SSD). Regarding the network configuration, all hosts are connected to one switch with a 1 Gigabit Ethernet network. Thus, latency introduced by the underlying communication network is neglected. For latency measurement all hosts are NTP time-synchronized.

On the Kafka configuration side the standard configuration has been applied with exception of the batch size, which has been set to 200 Bytes (the message size used in the test scenarios). Kafka API and broker system version 1.0.0 were used. Additionally in both tests a single topic with 1 partition has been used and the autocommit option has been activated at the consumer side (which is default) and acknowledgements at producer side are also activated. The messages, containing a sending timestamp in milliseconds and a sequence number, have a constant size of 200 Bytes and are de- and encoded using Kafkas string serializer.

A. *Latency Offset Evaluation*

In this test, each second a single message is sent from producer to consumer. In order to measure the latency, at consumer side, the receiving timestamp is compared to the sending timestamp in the message payload. Furthermore, the sequence numbers of packets are tracked to detect packet loss and reordering.

In a 30 minutes test 1800 messages have been sent from producer to receiver. The maximum latency value has been 10ms the minimum value 2ms. On average (mean) 3.53ms has been experienced. As can be seen from the ECDF graph in figure 3, the impressive majority (more than 95% of the values) lied below 6ms delay. During the tests it became apparent, that the latency for the very first message (and only the very first message) is around 200ms. This is due to connection establishment (e.g. TCP 3-way-handshake) and some Kafka related management operations. Thus, this value has been omitted for our analysis and is not included in the figure. Packet loss or reordering did not occur.

Altogether, the achieved performance is promisingly even for the usage in event-based communication common for many Smart Grid applications. Many latency-sensitive applications in Smart Grids (such as remote control of field devices) often require reaction times of 50ms or higher, and the experienced latency overhead is well below that value. Even more, with more application-specific system configurations, the experienced latency could be further decreased.

B. *Burst Messaging Test*

Instead of producing a single message every second in this test as many messages as possible are sent in burst mode. All other configurations remain unchanged compared to the first test (in particular batching is disabled). In this test scenario, basically the total number of successfully transmitted messages has been evaluated using the sequence numbers encoded in the messages.

During the 8 minute test period 1301170 messages were sent resulting to a rate of 2710 messages/sec. This equals to a goodput (only application data) of around 500 kilobyte/sec. Packet loss or reordering did not occur.

Further evaluations showed that the limitation of message rate stems from Kafka API configuration regarding autocommit at consumer side and acknowledgements for sent messages at producer side. Disabling both increased the massage rate to 28825 messages/sec (goodput: about 5500 kilobyte/sec). Thus, message rate and throughput increased by a factor of 10, roughly.

VII. CONCLUSION AND FUTURE WORK

While message-oriented middleware (MOM) is in general claimed to be a viable solution for usage in energy related applications, this work goes into details when MOM is applied for grid control based on inter- and intra-DSO flexibility-trading. This mechanism requires a messaging system fulfilling requirements regarding scalability, fault-tolerance, security, reliability and event-based data transfers.

A first result of our work is the proposal of Apache Kafka based messaging oriented middleware, embedded into a hierarchical system architecture consisting of various types of agents. It is shown, that several aspects of Kafka provide a beneficial basis in order to build a large-scale messaging solution respecting the postulated application specific requirements. Additionally we also provided details on a messaging API hiding much of the complexity of the messaging system and supporting implementers of agents.

As next step, this work will be extended by applying and validating the messaging solution in a pan-European testbed within the Callia project. Here, the main focus will be laid on fault-tolerance and performance evaluations in a large-scale WAN environment. In this large scale evaluation also different communication technologies, in particular LTE, 3G-PLC will be included.

Acknowledgment

Callia is part of the ERA-Net Smart Grids Plus initiative and carried out on behalf of the Austrian Federal Ministry for Transport, Innovation and Technology (BMVIT) with support from the European Union's Horizon 2020 research and innovation programme.

References

[1] M. Albano, L. L. Ferreira, L. M. Pinho, and A. R. Alkhawaja, "Message-oriented middleware for smart grids," Computer Standards & Interfaces, vol. 38, pp. 133–143, 2015.

[2] J. Kreps, N. Narkhede, J. Rao, and others, "Kafka: A distributed messaging system for log processing," in Proceedings of the NetDB, 2011, pp. 1–7. [Online]. Available: http://people.csail.mit.edu/matei/courses/2015/6.S897/readings/kafka.pdf

[3] B. Snyder, D. Bosnanac, and R. Davies, ActiveMQ in action. Manning Greenwich Conn., 2011, vol. 47.

[4] A. Videla and J. J. Williams, RabbitMQ in action: distributed messaging for everyone. Manning, 2012.

[5] M. Richards, R. Monson-Haefel, and D. A. Chappell, "Java Message Service: Creating Distributed Enterprise Applications," O'Reilly Media, Inc.", 2009.

[6] NSQ Docs 1.0.0-compat - A realtime distributed messaging platform." [Online]. Available: http://nsq.io/

[7] P. Hintjens, ZeroMQ: messaging for many applications. " O'Reilly Media, Inc.", 2013.

[8] Ferdinand von Tüllenburg, Georg Panholzer, Jia Lei Du et al., "An Agent-based Flexibility Trading Architecture with Scalable and Robust Messaging," in Proceedings of the 2017 IEEE International Conference on Smart Grid Communications (SmartGridComm)

[9] G. Pardo-Castellote, "Omg data-distribution service: Architectural overview," in Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on. IEEE, 2003, pp. 200–206.

[10] [10] A. R. Alkhawaja, L. L. Ferreira, and M. Albano, "Message oriented middleware with qos support for smart grids," in INForum 2012-Conference on Embedded Systems and Real Time., 2012.

[11] H. Yang, M. Kim, K. Karenos, F. Ye, and H. Lei, "Message-oriented middleware with qos awareness," in Service-Oriented Computing. Springer, 2009, pp. 331–345.

[12] J. Kramer, "Advanced message queuing protocol (amqp)," Linux Journal, vol. 2009, no. 187, p. 3, 2009.

[13] V. M. Ionescu, "The analysis of the performance of rabbitmq and activemq," in RoEduNet International Conference-Networking in Education and Research (RoEduNet NER), 2015 14th. IEEE, 2015, pp. 132–137.

[14] V. John and X. Liu, "A survey of distributed message broker queues," arXiv preprint arXiv:1704.00411, 2017.

[15] "Welcome to Apache Avro!" [Online]. Available: https://avro.apache.org/