An Easy-to-use, Scalable and Robust Messaging Solution for Smart Grid Research

Ferdinand von Tüllenburg, Jia Lei Du, Georg Panholzer

Salzburg Research Forschungsgesellschaft mbH, Salzburg, AUSTRIA,
email: {ferdinand.tuellenburg, jia.du, georg.panholzer}@salzburgresearch.at

Abstract: **Smart Grids are characterized by tight coupling and intertwining between the electrical system and information and communication technology. Due to this, application layer messaging systems are regularly required for many Smart Grid applications. Especially in research messaging solutions are setup from scratch. In this paper we propose a generic and easy to setup message oriented middleware (MOM) solution providing robust and scalable messaging.**

Keywords: **Smart Grid, Messaging API, Middleware**

I. Introduction

Future electrical power systems will be characterized by a new control paradigm: Decentralized controllable power sources such as batteries, wind generators, and PV systems on production side and controllable loads on consumption side will be constantly monitored and operated depending on the current grid state in order to increase overall efficiency and ensure power quality. Part of this development is the augmentation of the electrical system is with information and communication technology (ICT).

For the sake of data transfers between entities of the ICT subsystem usually a messaging solution is required providing an infrastructure to distribute messages correctly between instances also including definitions of message formats.

Especially for research projects in the field of Smart Grids, it is apparent that solutions for data transmission systems are redeveloped every time – a time consuming task when considering that the data transfer system is usually of minor priority. Due to this, an easy-to-deploy and (re-) usable messaging solution can be seen as a valuable contribution to Smart Grid research.

In this paper we introduce our messaging solution following the concept of a message-oriented middleware (MOM). As these features are regularly required in Smart Grid scenarios, the proposed solution provides robust, reliable, scalable and secure data transfers. All these without losing focus on ease of use and deployment, as well as applicability in various Smart Grid scenarios. In addition to this, we propose an application programming interface (API), which can be easily used and integrated in the communicating software components.

MOM solutions in general provide advantages for smart grid communication with respect to required communication capabilities (e. g. group communication) high scalability and high performance [1]. Additionally, one important beneficial aspect of using MOMs is that agents can focus on their key tasks of processing information, while the MOM handles issues regarding security, performance, scalability, reliability and robustness of sending and receiving messages.

The paper shows the application of the messaging solution in context of an agent-based flexibility trading application.

II. Related Work

In context of messaging systems for Smart Grid application especially solutions based on XMPP are often used [2]. Although, XMPP is a flexible solution also following a MOM approach, it has weaknesses with respect to ease of deployment and configuration as well as implementation especially with respect to required aspects such as reliability. Recently, with FIWARE, an open source platform is available which provides a large set of application programming interfaces (APIs) for a large variety of applications also providing a messaging solution for Smart Grids. However, the platform is extremely complex to setup and operated, thus lacking of ease-of-use.

In a more general sense, several MOM solutions are available. Those, however, have not been used widely in the field of Smart Grid applications. Particular types of MOMs use a (distributed) message broker as central hub for information interchange. Every sent message passes the messaging broker, which executes specific operations such as persisting, queuing or translating on each message. Especially with respect to reliability and robustness broker-based messaging has certain benefits such as life-time decoupling, state recovery, or guaranteed delivery [3].

Research and industry have developed several broker-based MOM systems providing similar services in general but have differences in their operational details and their specific focus. The solution ranges from research driven developments for certain use cases such as DoubleDecker [4] used for transmitting computer network status and monitoring data within a software-defined networking infrastructure, via solutions focusing mainly on high throughput such as ActiveMQ and RabbitMQ or lightweight and easy-to-use solutions such as NSQ [5] or NATS [6] towards totally cloud based solutions like Amazon SNS+SQS or Microsofts Azure Platform. While the envisioned flexibility trading messaging solution would be potentially realizable with any of the MOMs, it was decided to use Apache Kafka [7]. The main reason were detailed documentation, the active development community, its claimed reliability and its use in productive environments such as by companies like LinkedIn.

III. Agent-based Flexibility Trading

The context in which the proposed messaging solution is show cased is a flexibility coordination scenario. Here, power equalization is achieved by orchestrating flexibilities of individual assets - particularly batteries, photovoltaics, and industrial loads. This means that assets adapt their production or consumption either by pre- or postponing or variation of amount. The flexibility coordination is implemented in a market-based approach, where flexibility offers are submitted towards are central market platform, where an optimal assignment of flexibilities is evaluated and corresponding control signals are sent towards flexibility suppliers.

The flexibility trading architecture is designed as hierarchical multi-agent system. Certain types of agents are situated at one of four logical layers fulfilling a specific task at this layer.

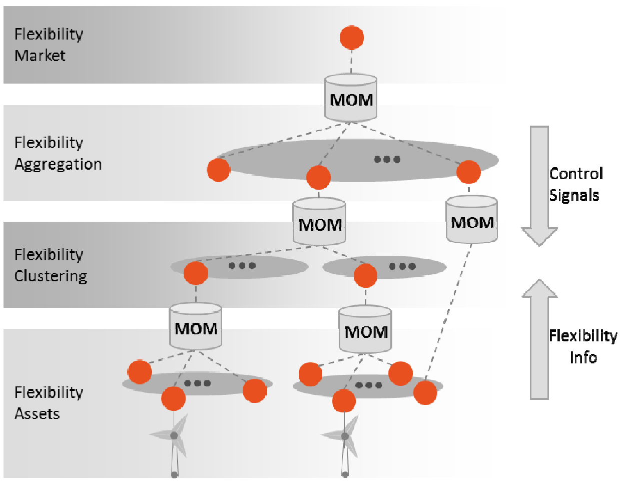

Figure 1 Overview of the scalability-focused flexibility trading hierarchy including MOM instances.

Figure 1 shows the layered agent-architecture. 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 at the market platform from the transmitted bids with respect to the current grid state.

From perspective of the information flow, in direction 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.

The deployment of 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 Kafka instances are provided and individually configured by either a cluster owner (e. g. a facility operator), an aggregator participating in flexibility trading and the market platform operator. This approach takes account of 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 number of connected agents at each Kafka deployment is limited to a small amount compared to a centralized solution. Both aspects are especially important for large scale (e. g. pan-European) deployments where thousands of agents participate in flexibility trading.

IV. Kafka-based Message Brokering

The MOM implemented in the proposed messaging solution is based on Apache Kafka. This section describes principle features of Apache Kafka justifying the selection as basis for the messaging solution.

Apache Kafka is designed as a distributed streaming platform following the principle of a broker-based message oriented middleware. Kafka's goal is to provide a means for high performance processing of continual sequence of input data (data packets) and its main design goals are reliable data transfer, fast processing, scalability, and fault tolerance [7]. These are important features for a Smart Grid messaging solution with respect to potentially many participating systems and overall criticality of such systems.

From a coarse-grained view, a Kafka-based distributed system consists of three principal components: The first component, the message producer, is creating streams of data that are read and/or processed by one or more message consumers (second component). The third main component is the Kafka messaging broker, which is mainly responsible for storing all messages until they are received by the message consumer. A Kafka message consists basically of a key/value pair for information encoding, a time stamp, and addressing information given by the core concept of message topics. A topic basically provides a name for a category of messages with a certain type. Using that name a message producer can publish a message of that type. On the other end, consumers can subscribe to messages of that type also using the topic name. Each topic may have multiple consumers.

Several aspects specific to the design of Kafka are important to achieve the requirements specifically stated for the flexibility-trading application. With respect to scalability of the system Kafka implements an efficiency-focused way of message handling by introducing the concept of partitions.

In order to provide high performance when messages are published or consumed, Kafka uses so-called partitions. 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. Each partition is usually managed by one instance of a Kafka cluster potentially running at a dedicated host. In that way, write and read performance of a topic is substantially increased through allowing parallel writes and reads by multiple consumers and / or producers. 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, basically, a consumer polls a topic and retrieves the messages of any of the partitions. Furthermore, in order to provide a means of load balancing and parallel processing at receiver side, consumers may be organized in consumer groups, which lead to direct mapping between consumers within that group and certain partitions: Each consumer of the group retrieves the messages of one specific partition. Parallel reads and writes minimize the additional latency induced by message brokering in general. Using that approaches Kafka was able to achieve a throughput of 500 K Messages/sec [7].

In terms of reliable data transfers, Kafka uses multiple approaches. First, TCP is used for a reliable message transfer between producers or consumers and the message broker, which ensures that messages are not lost during transmission. Additionally, Kafka provides an at-least-once guarantee for message delivery ensuring that each message queued in a topic is delivered to its recipient one or more times. For each consumer a pointer is maintained indicating which message has to be received next by a certain consumer. The pointer is only set to the next message, if reception of the message has been explicitly acknowledged by the message consumer.

With respect to fault-tolerance of the Kafka messaging system, each partition might have replications on other Kafka instances providing a means of redundancy. One of the replicas is considered as leader, where messages are stored first and afterwards replicated to other instances. Only after the message has been hard-drive-persisted at each replication, subscribed consumers can pull the message. This message persistence also provides a means for fault-tolerance in case of system failure of an Kafka instance. Together with a configurable retention time, specifying how long messages are stored in the Kafka system, this also provides a means of system recovery for message consumers, which then have the possibility to restore their internal state after a system failure by polling old messages from the respective topics.

In terms of security, Kafka provides TLS-secured connections between consumers/producers and the broker systems, and additionally access control lists (ACLs) specifying access rights (read, write) a consumer or producer has for each topic.

V. Configuration and Usage of the MOM Solution

Extending the proposed Kafka deployment, an easy-to-use messaging middleware client API for inter-agent data transfers was developed, aimed to be used by software component implementers. Currently, the API is available for Java and Python. Additionally, an efficient message serialization system based on Apache Avro [8] is included.

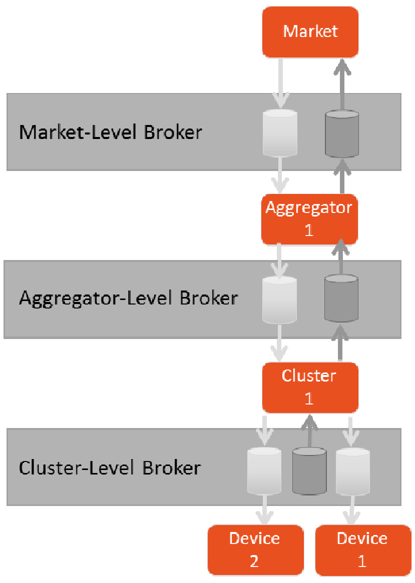

Figure 2: Topic configuration for the flexibility trading use case.

The main design goals of the messaging client API were the provisioning of a high-level abstraction of Kafka messaging details such as creating and maintaining Kafka connections, topic and partition management, TLS key management as well as message encoding and serialization. This allows agent implementers to focus on developing agent logic despite of concentrating on messaging details. While the API and configuration is suitable for any arbitrary topology, we showcase the setup for the layered agent hierarchy as used for the flex trading use case. The corresponding topic configuration is shown in Figure 2.

Essentially, each agent uses one topic for incoming messages from lower layers (flexibility information), while top-down messages (control information) is delivered using private topics only readable by the addressed agents. This is chosen because of data security reasons but also in order to minimize the number of actively maintained connections for each agent (maximum of three in this case).

Before an agent is able to send or receive messages it must be registered at the Kafka broker including exchange of necessary security credentials. Within the agent hierarchy, each agent has connections to one or two Kafka brokers - one broker for upstream data transfers and/or one for downstream. The messaging API provides the functionality to correctly connect an agent to the Kafka messaging broker. For this, however, a minimal set of information needs to be provided in advance for each connection an agent has to another agent in form of a simple configuration file which is given in Listing 1.

```
connection.name=aggreCon
agent.id=cluster-1
broker.address=192.168.100.135:9093
topic.in=cluster-1-in
security=TLS
```

Listing 1: Configuration file specifying settings for an agent connecting another agent

Besides an unique agent identifier (`agent.id`) and a connection name (`connection.name`), the addressing information of the Kafka instance (`broker.address`), and the security configuration needs to be specified (`security`).

The API provides simple methods for sending (`send(Message)`) and or receiving messages (`receive()`) from other agents. The API user must not be aware of correct topic names and/or partitions or acknowledging received messages. The usage of the API is shown for cluster agent `cluster-1` in Listing 2 for the Java version:

```
//…
//instantiation of a connection object
Connection aggreCon = new Con("aggreCon");
//…
//create a MessageObject
Message request = new Message("Hello");
//reliably send message to the aggregator
aggregatorCon.send(msgObj);
//receive all messages from the aggregator
Message[] answers = aggreCon.receive();
```

Listing2: Instantiation and usage of a Connection object.

First, a Connection object is instantiated describing a connection to a particular broker instance as specified in the configuration file (which is automatically read when instantiated). Second, a message object is created. Third, the message is sent using the `send()` method provided by the API by specifying the addressee of the message. After that the user can be sure, that the message will be reliably delivered to the receiver. Calling the `receive()` functions fetches all messages stored in the corresponding topic (`topic.in`). Resolution of actual topic names as well as making sure that messages are sent and delivered is hidden by the API.

While configuration and usage of the messaging API seems to be simple, some remarks has to be made:

If (as in this case) private topics are used, 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.

With respect to scalability and efficiency of data transfers, Apache Avro \cite{avro} 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 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.

VII. Conclusion and Future Work

While message-oriented middleware (MOM) is in general claimed to be a viable solution to be used in energy related applications, this work goes into details when MOM is applied for power grid control in context of a flexibility-trading application. This mechanism requires a messaging system fulfilling requirements on scalability, fault-tolerance, and secure and reliable 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 mainly conceptual work will be extended by applying and validating the messaging solution in a pan-European testbed. Here, the main focus will be laid on fault-tolerance and performance evaluations with respect to latency or throughput. In this context we plan to investigate impacts of different Kafka deployment scenarios in comparison to the currently chosen multilayer deployment. Furthermore, we will extend this work by implementing end-to-end security including reliable key deployment for agent-to-agent communication. As secure data transfers are usually required by many energy-related applications, this already goes in line with a final goal of our work to develop a general purpose messaging solution not solely for flexibility trading but for a broad variety of distributed applications related to energy-systems.

Last, but not least, the solution will be complemented with a configuration tool, automatically creating connection configuration files, which, in turn, might be automatically distributed to the agents via a (also to be developed) agent bootstrapping support system.

References

[1] Michele Albano, Luis Lino Ferreira, Luis Miguel Pinho, and Abdel Rahman Alkhawaja. "Message-oriented middleware for smart grids," *Computer Standards & Interfaces*, vol. 38, Feb. 2015), 133–143.

[2] Peter Saint-Andre, "RFC6120 - Extensible Messaging and Presence Protocoll (XMPP): Core." *Internet Engineering Task Force (IETF)*, Mar-2011.

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

[4] Wolfgang John, Catalin Meirosu, Bertrand Pechenot, Pontus Skoldstrom, Per Kreuger, and Rebecca Steinert. "Scalable Software Defined Monitoring for Service Provider DevOps," IEEE, Oct. 2015, pp. 61–66.

[5] [n. d.]. NSQ Docs 1.0.0-compat, *A realtime distributed messaging platform*. http://nsq.io/.

[6] [n. d.]. NATS - Documentation. ([n. d.]). https://nats.io/.

[7] Jay Kreps, Neha Narkhede, Jun Rao, and others, "Kafka: A distributed messaging system for log processing," *Proceedings of the NetDB*, 2015 pp. 1–7.

[8] [n. d.]. "Welcome to Apache Avro!" https://avro.apache.org/.