Università degli Studi di Camerino
MSc in Computer Science (LM-18)
Data Analytics 2020-2021

Alessandro Zallocco
July 25, 2021

Abstract

Apache Qpid is an open-source messaging system which implements AMQP and supports many languages and platforms. In this project, Apache Qpid and its functionalities have been investigated in relation to centralized cloud computing and edge computing in Internet of Things (IoT). From this analysis, a Maven project has been developed in order to present some of the Apache Qpid components and features.

1 Introduction

Internet of Things (IoT) intends to revolutionize our life by connecting everything around us with each other, affecting almost all aspects of our life (e.g., homes, offices, healthcare, transportation, industries...). In this context, IoT or intelligent devices (e.g., smart mobiles, smart watches, smart security system, medical sensors, fitness trackers...) are entities with embedded computational capability that can produce data, sources of information that need to be processed. In this project, Apache Qpid has been studied in order to discover its use cases in support of both edge and cloud computing scenarios and develop a prototype for showing the lessons learned from this research.

2 Background

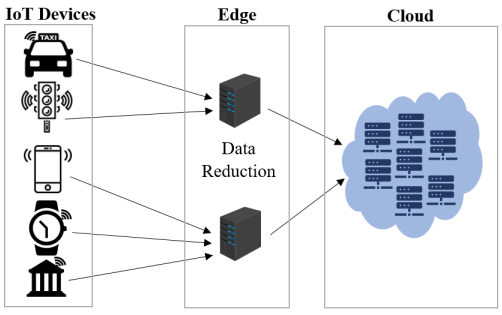

Figure 1: IoT scenario

IoT devices are connected to local gateways, where generated data is captured and the information is sent to a remote cloud platform in which we implement standard business analytic applications. As the number of devices continues to increase, a major issue is to generate useful information through computation. That said, data provided by this type of devices can be sent to a remote cloud environment after being collected and also pre-analyzed in a prior edge computation layer, as shown in Fig. 1. In a cloud environment, a bigger computational power can be used and a huge amount of data can be handled with different analytics frameworks like Apache Spark, a fast and general engine for large scale data processing [2]. In particular, Apache Spark is the central core that provides basic analytics and all the functionalities to upload data, transform data and produce some output. Additionally, other vertical modules like Spark Streaming to implement analytics in different ways are defined.

3 Apache Qpid

Modern large-scale applications are often built as distributed network applications, with parts of the application in distinct processes and distinct parts of the world. These parts need a way to communicate and they must be able to tolerate failures in order to work together as one reliable application. In these circumstances, a store-and-forward messaging system provides efficient, reliable communication since message brokers take responsibility for ensuring messages reach their destination, even if the destination is temporarily out of reach, and messaging APIs manage acknowledgments so that no messages are dropped in transit. The Qpid project [1] offers messaging APIs and message brokers for use in diverse applications as well as core libraries based on AMQP, the first open standard wire protocol for reliably sending and receiving messages.

Figure 2: Apache Qpid

Apache Qpid proposes two kinds of components:

- Messaging APIs are a software tool for efficient, high-level interprocess communication within applications.
  - Qpid Proton is a toolkit allowing any application to speak AMQP. It is used by other Qpid components to implement AMQP 1.0 protocol support.
  - Qpid JMS is an AMQP-fluent Java Message Service implementation.
  - Qpid Messaging API is a connection-oriented messaging API that supports many languages.
- Messaging servers are message-transfer intermediaries that provide additional behaviors such as store-and-forward for improved reliability. They are full-featured message-oriented middleware brokers that offer specialized queueing behaviors, message persistence, and manageability.
  - Broker-J is a pure-Java AMQP message broker.
  - C++ broker is a native-code AMQP message broker.
  - Dispatch router is an AMQP router for scalable messaging interconnect between any AMQP endpoints, whether they be clients, brokers or other AMQP-enabled services.

The two components that have been mainly explored during the making of this project were Broker-J and Qpid JMS.

3.1 Broker-J

The Apache Qpid Broker-J is a powerful open-source message broker that implements all versions of AMQP. It is an 100% Java implementation as such it can be used on any operating system supporting Java 1.8 or higher [4].

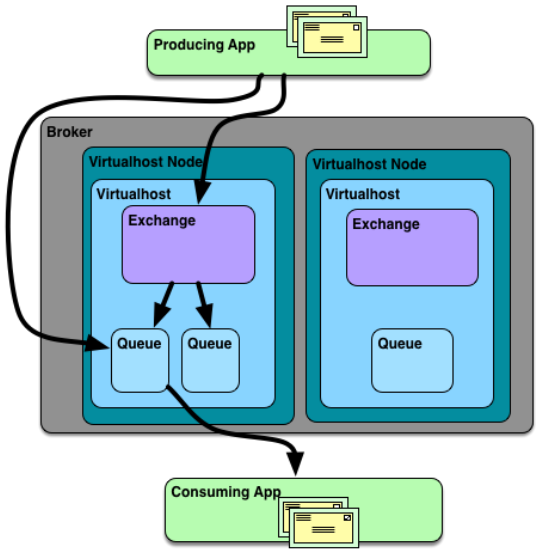

Figure 3: Broker-J entities

As shown in Fig. 3, the *Broker* comprises a number of entities. The most important entity is the *Virtualhost*, an independent container in which messaging is performed. A *Virtualhost* exists in a container called *Virtualhost Node* and each *Virtualhost Node* has exactly one *Virtualhost*, for which it provides an initial configuration. A *Virtualhost* is identified by a name which must be unique broker-wide and it used by the clients to identify the *Virtualhost* to which they wish to connect when they connect (the messaging that goes on within one *Virtualhost* is independent of any messaging that goes on in another *Virtualhost*). The *Virtualhost* comprises a number of entities, too; *Exchanges* is a named entity which receives messages from producers and routes them to matching *Queues*, another type of named entities that hold messages for delivery to consumer applications. When using AMQP 1.0, producers may route messages via *Exchanges* or direct to *Queues*. Moreover, *Connections* represent a live connection to the *Virtualhost* from a messaging client, a *Session* represents a context for the production or consumption of messages (a *Connection* can have many *Session*) and a *Consumer* represents a live consumer that is attached to *Queue*. Finally, the *Broker*, a *Virtualhost Node* and a *Virtualhost* are backed by storage, used to record the durable entities that exist beneath the *Broker* and the *Virtualhost Node*, and to store the messages, respectively.

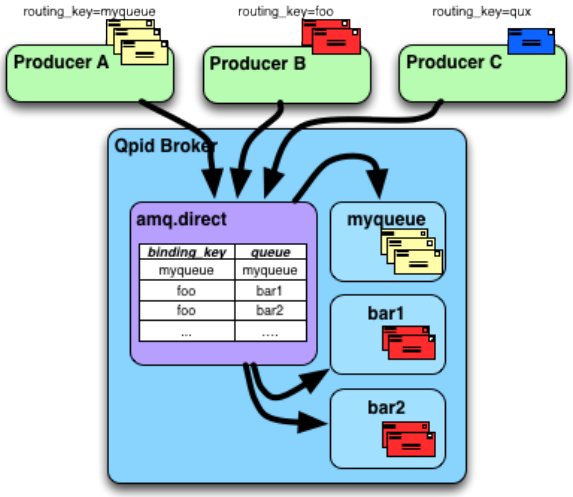

Figure 4: Direct exchange type example

The supported *Exchange* types are: *Direct*, *Topic*, *Fanout* and *Headers*. They are declared by each *Virtualhost* (amq.direct, amq.topic, amq.fanout and amq.match). The Fig. 4 illustrates the operation of *Direct Exchange* type, that routes messages to *Queues* based on an exact match between the routing key of the message and the binding key used to bind the *Queue* to the *Exchange*. Furthermore, the *Broker* supports four different queue types, each with different delivery semantics: *Standard* (FIFO), *Priority*, *Sorted* and *Last Value Queue*.
HTTP is the primary channel for management; the HTTP

Management plugin provides a HTTP based API for monitoring and control of the Broker through two interfaces: Web Management Console and REST API. The Web Management Console is a rich web based interface for the management of the Broker, in which it is possible to control different aspects, including add, remove and monitor queues or virtualhosts. Providing the HTTP Management Plugin is in its default configuration, the Web Management Console can be accessed by pointing a browser at the following URL: *http://myhost.mydomain.com:8080*. The Console will prompt you to login using a username and password. After you have logged on you will see a screen similar to the following (Fig. 5), in which all the entities of the Broker can be managed.

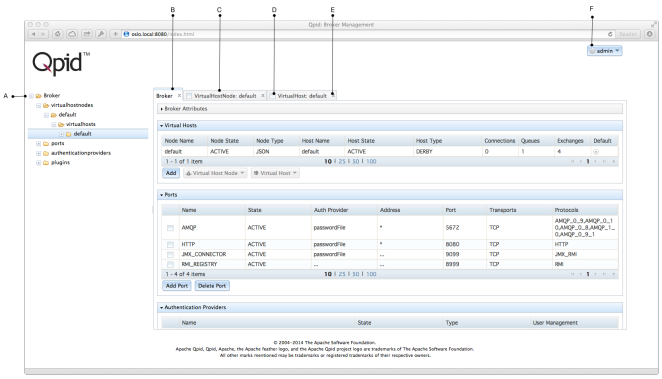

Figure 5: Web Management Console

3.2 Qpid JMS

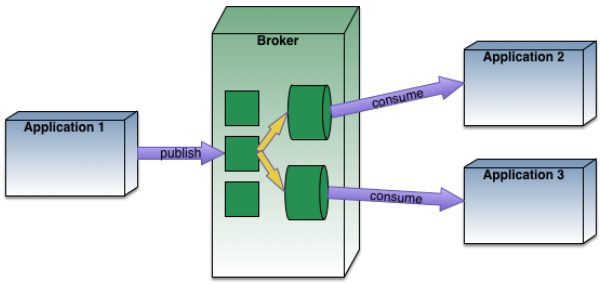

Figure 6: Qpid JMS overview

Qpid JMS (Fig. 6) is an AMQP 1.0 Java Message Service 2.0 client built using Qpid Proton [5]. A *ConnectionFactory* allows an application to create a *Connection*, an open communication channel between application and Broker. Each connection utilises a single TCP/IP connection between the process of the application and the process of the Broker (the underlying TCP/IP connection remains open for the lifetime of the JMS connection). The application obtains the *ConnectionFactory* from an *InitialContext*. The *InitialContext* is itself obtained from an *InitialContextFactory*. Additionally, the Client defines JNDI properties that can be used to specify JMS *Connections* and *Destinations*.

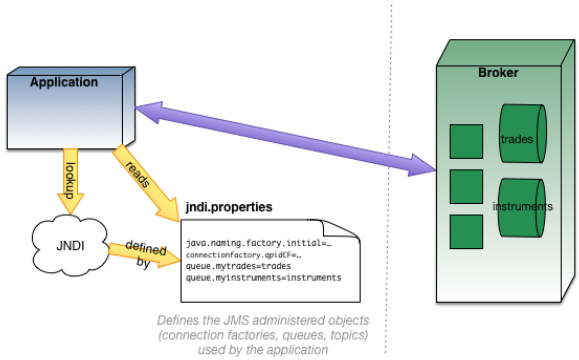

Figure 7: JNDI overview

As it is shown in Fig. 7, the Client provides a single implementation of the *InitialContextFactory* in class *org.apache.qpid.jms.jndi.JmsInitialContextFactory*. Moreover, a *Session* object is a single-threaded context for producing and consuming messages. *Session* objects are created from the *Connection* and the number of sessions open per connection at any one time is limited. A *MessageProducer* sends a message to an *Exchange* (within the Broker) that routes the message to zero or more *Queues*. Routing is performed according to rules expressed as bindings between the *Exchange* and *Queues* and a routing key included with each message. *MessageConsumer* objects are created from the *Session* and a *MessageConsumer* receives messages from a *Queue*. Finally, a *Destination* is either a *Queue* or *Topic*. In the Client a *Destination* encapsulates a Binding URL, but applications can also refer to JMS administered objects declared in the JNDI properties file with the *queue.* and *topic.* prefix to create *Queue* and *Topic* objects respectively.

4 Demo

The previously described concepts have been used in a prototype project [3], in which Apache Qpid, in particular Qpid JMS and Broker-J, has been used in relation to edge and cloud computing. Initially, the installation of Broker-J on Windows has been performed choosing a directory for the Qpid JARs and configuration files (more generally for the extraction of the broker downloaded package). Qpid also requires a work directory used for the default location of the Qpid log file and for the storage of persistent messages. In this case, it is important to set the environment variable permanently via the Advanced System Settings in the Control Panel. Once the Broker is started, subsequent management is performed using the Management interfaces. The configuration for each component is stored as an entry in the broker configuration store, currently implemented as a JSON file. Broker startup involves two configuration related items, the *Initial Configuration* and the Configuration Store. When the broker is started, if a Configuration Store does not exist at the current store location then one will be initialised with the current *Initial Configuration*. Subsequent broker restarts will use the

existing configuration store and ignore the contents of the *Initial Configuration*.

5 Conclusion

Apache Qpid makes messaging tools that speak AMQP and support many languages and platforms. AMQP is an open internet protocol for reliably sending and receiving messages. It makes it possible for everyone to build a diverse, coherent messaging ecosystem. In this project, the possible roles of Apache Qpid in IoT scenarios have been investigated and a presentation for its components, as Broker-J and Qpid JMS, has been proposed. Finally, an example project showing learned Qpid concepts has been develop in order to expose offered opportunities supporting data analytics.

References

[1] http://qpid.apache.org/index.html

[2] https://spark.apache.org/docs/latest/index.html

[3] https://github.com/alesszall/DA21

[4] https://qpid.apache.org/releases/qpid-broker-j-8.0.5/book/index.html

[5] https://docs.oracle.com/javaee/7/api/javax/jms/package-summary.html