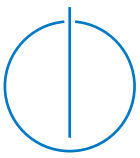
FAKULTÄT FÜR INFORMATIK

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Shared Dictionary Compression within a Distributed Publish/Subscribe System

Manit Kumar



FAKULTÄT FÜR INFORMATIK

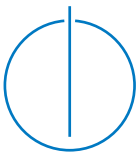
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Shared Dictionary Compression within a Distributed Publish/Subscribe System

Kompression auf Basis verteilter Wörterbücher in Publish/Subcribe Systemen

Author:	Manit Kumar
Supervisor:	Prof. Dr. rer. pol. Hans-Arno Jacobsen
Advisor:	Christoph Doblander
Submission Date:	TODO: Submission date



I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, TODO: Submission date Manit Kumar

Acknowledgments

I would like to express my sincere appreciation to my advisor Christoph Doblander, for his support and guidance throughout my thesis. His valuable knowledge helped to take this work into new directions.

I would also like to thank Prof. Dr. Hans-Arno Jacobsen, for not only supervising my thesis but also for all the lessons I have learned from him during the two years of my master's study.

Last but not the least, I would like to thank all the faculties at my department for their support and guidance.

Abstract

Publish-subscribe middleware paradigm is a scalable solution for large scale distributed systems. Due to its asynchronous nature, it enables a simple interface for clients in addition to the high degree of decoupling between the publishers and subscribers. To achieve high-throughput, the current state of the art implements a Shared Dictionary Compression for Publish-subscribe (SSPS). However, the present implementation of SSPS is on a centralized broker.

In a Simple Shared Dictionary Compression for Publish-subscribe, the component Sampling Broker (SB) is responsible for the creation of the dictionaries, observing the compression ratio and maintaining the dictionary. However one of the major drawback is that the broker is a centralized entity and hence it is prone to all the drawbacks of a centralized system like single point failure, limited scalability and so on.

In this thesis, we implement the SSPS on a distributed broker called Apache ActiveMQ Artemis. An extension is developed for Apache ActiveMQ Artemis which overcomes the disadvantages of the current SSPS. The working of SSPS is demonstrated by an Android application.

Contents

1 Introduction

Publish-subscribe is a messaging pattern where the senders of the messages are called *publishers*, and the receivers of the messages are called *subscribers*. The messages from publishers are characterized into classes. The subscribers receive messages from one or more classes to which they express their interest. Publish-subscribe pattern has a widespread adoption in distributed systems due to a high degree of decoupling. This is because the publishers and subscribers have no knowledge of each other. The co-ordination is done by an entity called the broker. Both the publishers and subscribers only communicate with the broker. The publisher publishes data without any information regarding the identity, location or the number of subscribers. Similarly, the subscribers receive the data without any information regarding the publishers. The other main advantage of this pattern is scalability. Publish-subscribe pattern provides the room for better scalability compared to traditional client-server due to message caching, parallel operation, network-based routing, etc. The earliest mention of publish-subscribe systems was the *news* subsystem of the Isis Toolkit [15].

One of the popular adoptions of this paradigm is the Topic-based publish-subscribe pattern. In this approach there exists a named logical channel called *topics*. All the subscribers will receive the same content from a topic to which the publishers publish the content.

To leverage the topic-based publish-subscribe pattern further in scenarios where bandwidth savings is important a Shared Dictionary Compression has been introduced recently [20]. This fairly new technique in the realm of publish-subscribe pattern aims at reducing bandwidth and in turn reducing costs. It allows the use of collaborative applications even in areas, where bandwidth is sparse.

1.1 Shared Dictionary Compression

Shared Dictionary Compression(SDC) is the idea of using dictionaries for achieving compression of data transmitted between the sender and receiver. The dictionaries can either be stored locally, or it can be uploaded from some source and then cached. One of the paper[10] that uses the dictionary based compression for English text using a small dictionary mentions compression ratio of 60%-70%. This idea is further explored by J. Bentley and D. Mcllroy in [11] and subsequently extended in [12].

The core of the Shared Dictionary Compression is the generation of the dictionary. The generation of the dictionary is done by mining the long common strings from the context. FemtoZip [24] is an open-source solution by Garrick Toubassi [49].

This idea of SDC was introduced in the realm of publish-subscribe pattern in the work Simple Shared Dictionary Compression for Publisher-Subscriber (SSPS) model [20]. Here the similarity between the notifications is leveraged to improve compression ratios. The SDC here is a combination of a dictionary and multiple passes of Huffman Coding. Using this method, the references to the dictionary are represented very efficiently.

1.2 Problem

Centralized systems as the name suggests are systems where one primary system manages all computing resources. In spite of the benefits of centralized systems such as small capital and operational cost (minimum hardware) they are often faced with problems such as single point failure, limited scalability, computational bottleneck, fault tolerance and so on. Thus centralized systems are unsuitable for many large real world applications.

In the Simple Shared Dictionary Compression for Publisher-Subscriber (SSPS) model, the entity called Sampling Broker(SB) is responsible for sampling notifications to create dictionaries, maintaining the dictionary and spreading the dictionary to the publishers and subscribers. To the best of our knowledge currently the only implementation of SSPS exists on one centralized broker called Moquette [41]. However, one of the major drawbacks is that this sampling broker is a centralized entity and hence it is prone to all the disadvantages of a centralized system.

Figure 1.1 depicts the current state of the art implementation of the Sampling Broker of

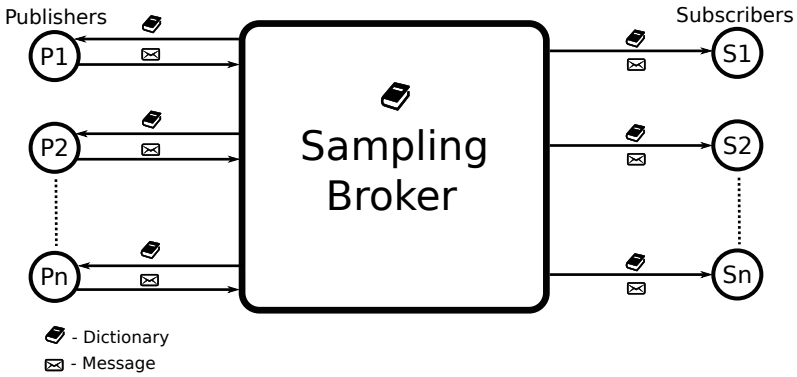


Figure 1.1: Sampling Broker in SSPS model

the SSPS model. As the number of publishers and subscribers increase, there would be a computational bottleneck. Also in the case of failure, the entire SSPS model would collapse, which is not desirable in the real world. Thus, there is a need to overcome the problem of this centralized entity.

1.3 Goals and Contribution

The core of the problem is that the Sampling Broker is a centralized entity. In general, the problem of a centralized system is solved by replacing it with a distributed system. Hence, in this case, the idea is to replace the centralized broker with a distributed broker. To the best of our knowledge currently there exist no implementation of SSPS on any of the available distributed brokers.

There are several open-source distributed brokers like Apache ActiveMQ Artemis, Apache Kafka and so on. The first part of this thesis would be to study the different open-source distributed brokers and select one among them for implementing the SSPS. Secondly, once an appropriate broker has been chosen, the architecture of the broker needs to be understood in order to extend it to incorporate the SSPS. Then, we design and implement a prototypical extension for the SSPS. Figure 1.2 depicts the design of the Sampling Broker in the case of a distributed broker. In comparison to Figure 1.1 it can be seen that now there are several instances of the sampling broker instead of only one instance.

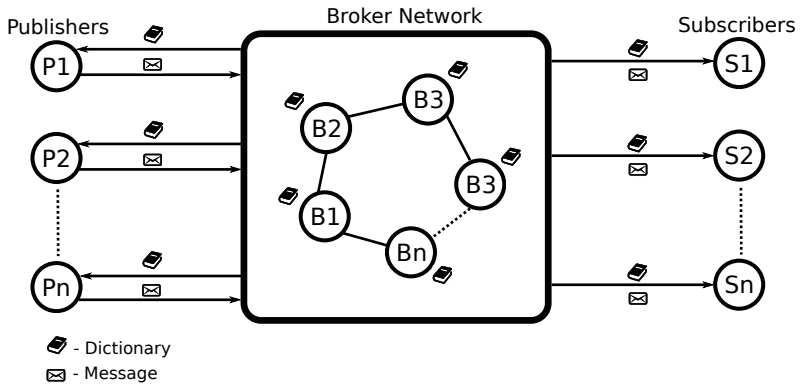


Figure 1.2: SSPS in a distributed broker

Once the implementation is complete, we test our prototypical implementation to get performance aspects such as throughput, CPU and memory usage and behavioral aspects such as the effect of SSPS in the case of failover. Finally, we create an android application to demonstrate the working of SSPS. In brief, the Android application would provide the provision to subscribe to a set of topics. Each topic would correspond to messages being received by the Android application in different formats such as JSON, XML, and CSV. The messages would contain location coordinates which will be displayed on a map. The Android application would also demonstrate the throughput comparison with and without the use of SSPS.

1.4 Thesis Outline

The information in this thesis document is grouped into 9 chapters. It is organized as follows : Chapter 1 gives the introduction to the thesis. It illustrates the problem and the goal of this thesis. Chapter 2 starts with covering the background and fundamental concepts that are necessary for the reader in order to understand the rest of the thesis. Chapter 3 enumerates the different distributed brokers. It shows the comparison between the different brokers and subsequently the selection of one of the brokers for our implementation of the SSPS model. Chapter 4 explains the broker chosen in chapter 3 to extend the SSPS model. It describes the major aspects that are required in order to implement the SSPS model. Chapter 5 deals with the android application which demonstrates the SSPS model. Chapter 7 deals with results for our prototypical implementation of the SSPS model on the distributed broker. Finally, in chapter 8, we

provide some guidance and insights for future work followed by a brief conclusion in chapter 9 summarizing the work we have done in this thesis.

2 Background

This chapter provides brief knowledge of the fundamental concepts that are required to understand the next sections of this thesis. It is divided into 8 sections. It starts with the first section explaining concept of middleware and its use cases. The second section describes message brokers. Third section gives a brief insight about clusters which is used in testing of the work done in this thesis. Then the fourth section deals with publish subscribe system and discuss the different types of publish subscribe systems and their differences. Following that the fifth section discusses Java Messaging System (JMS). Sixth section deals with the MQTT protocol which is used with SDC. The seventh section deals with Shared Dictionary Compression in a topic based publish subscribe system. Finally the chapter ends with the discussion of the related work done in this area.

2.1 Middleware

Middleware is a software layer between software applications and operating systems that provide services to the applications apart from the ones available from the operating systems [14]. For software developers to focus on the specific purpose of the application, middleware makes implementation of input/output and communication easier.

In the context of distributed systems, middleware is responsible for communication and management of data among the nodes. It simplifies various systems that support application development and delivery like web servers, messaging systems, application servers and other similar systems. The primary purpose of a middleware system is to provide the abstraction for the complex interaction that takes between heterogeneous nodes in a distributed system. The complexity of services such as concurrency, location, naming and service discovery are hidden from the application layer. It does so by providing a common Application Programming Interface. Figure 2.1 depicts a typical high-level design of the involvement of middleware in a distributed systems scenario.

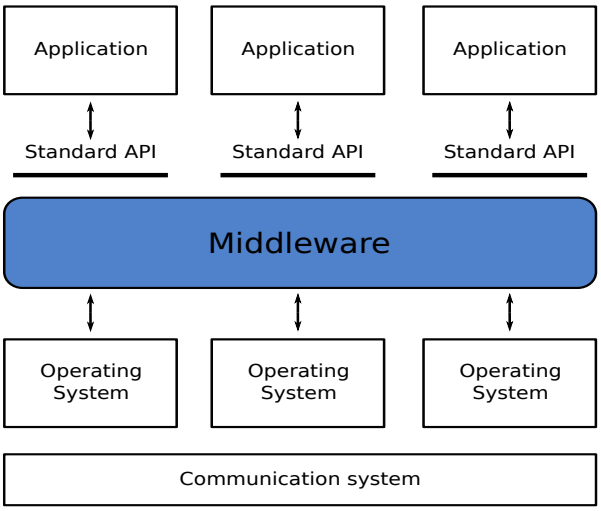


Figure 2.1: Middleware

There are many types of middleware [54]. Few of them are listed below:

- **Message Oriented Middleware:** These are the class of middleware systems that perform routing and transformation of messages. These are asynchronous in their nature of operation. A good example is integration brokers.
- **Remote Procedure Call (RPC) Middleware:** As the name suggests these class of middleware is used to calling procedures on remote systems. The interaction between the caller and callee systems can be synchronous or asynchronous.
- **Database Middleware:** These middleware allows direct interaction with databases. Extract, Transform, and Load (ETL) tools like Pentaho come under this category
- **Embedded Middleware:** These type of middleware allows embedded applications to communicate with real-time operating systems.
- **Portals:** These middleware deal with the interaction between the front end and the back end services.

2.2 Message Brokers

In the context of distributed systems, a message broker is an entity that coordinates the communication between a sender and a receiver. Message brokers are the core of Message Oriented Middleware. It enables the decoupling of sender and receiver by not having to keep the knowledge of each other [37]. It increases the modularity of an application. 2.2 depicts the role of message broker, where it acts as co-ordinator between the publishers and subscribers.

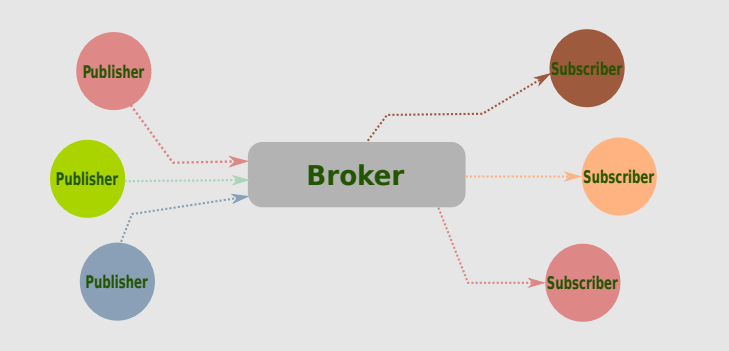


Figure 2.2: Message Broker

While the primary function of a message broker is to decouple a sender and receiver, it can also perform other activities [52] such as:

- Messaging routing to one or more destinations
- Message transformation to an alternative representation
- Aggregating messages and delivering them to the receiver. Further receiving acknowledgments and composing them into a single response to deliver back to the sender.
- Routing messages in content and topic-based publish-subscribe pattern
- Provides guarantees regarding message delivery and transaction management.

2.3 Clusters

Clusters are regarded as a subset of distributed systems. Unlike distributed systems the nodes in a cluster are mostly homogeneous having the same hardware and operating system. They are dedicated to performing tasks that are well defined by acting as one single entity. The brief explanation of clusters may sound similar to distributed systems and hence to distinguish between the two classes the table 2.1 lists some of the key differences between clusters and distributed systems.

Table 2.1: Clusters vs. Distributed systems

Aspect	Clusters	Distributed systems
Structure	Homogeneous	Heterogeneous
Scale	Small scale	Medium or large scale
Task	Specialized	General
Security	Nodes trust each other	Nodes do not trust each other

2.3.1 Cluster Attributes

Clusters can be used for various tasks ranging from general purpose tasks to resource intensive scientific computations. The two important attributes of clusters are load balancing and high availability.

- **Load-balancing:** This refers to the balancing load of computation among different nodes of the cluster. The balancing technique can range from simple round-robin fashion to any sophisticated algorithms depending on the nature of the task that needs to be performed. Figure 2.3 depicts a cluster setup in which a dispatcher sends the web requests to the servers ensuring that the load is evenly distributed among them.

- **High-availability:** This refers to providing fault tolerance eliminating the single point of failure. This is done by having redundant nodes which take over the nodes that fail during the regular operation. They are also referred as failover clusters. Figure 2.4 depicts a file service cluster setup where the active machine serves files to the clients and the standby machine acts as a backup. The backup

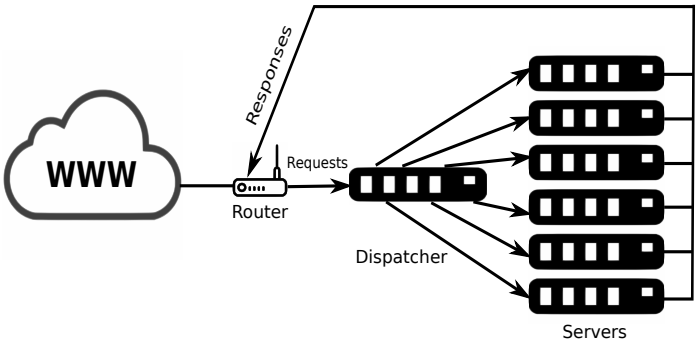


Figure 2.3: Load-balancing cluster Example

keeps its copy of files updated concerning the active machine. In case the active machine fails, the backup machines realizes this and takes over the active server and continues to serve the files to the clients. Thus even in a case of failure, the service is not shutdown completely.

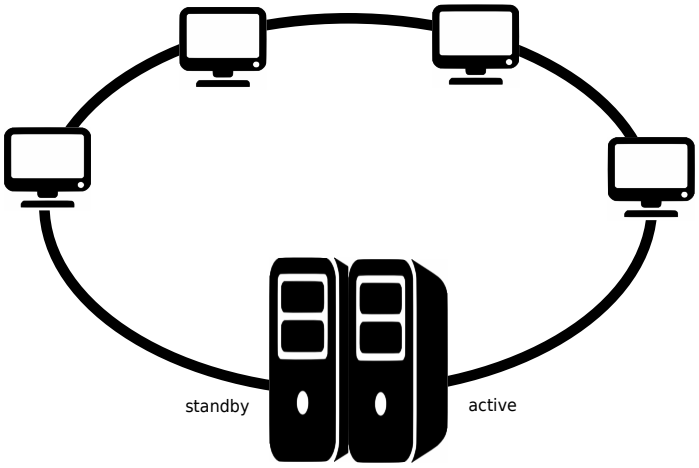


Figure 2.4: High-availability cluster Example

2.4 Publish-subscribe systems

Publish-subscribe is an asynchronous communication pattern where the senders of the messages or producers of the content are called *publishers*, and the receivers of the messages or content consumers are called *subscribers*. The subscribers express their interest in being notified and receiving content that matches their interest whenever matching contents are published [45].

The two primary advantages of the publish-subscribe model are loose coupling and scalability.

Loose Coupling

The publish-subscribe pattern has a widespread adoption in distributed systems due to a high degree of decoupling. This is because the publishers and subscribers have no knowledge of each other [16]. The co-ordination is done by an entity called the broker. Both the publishers and subscribers only communicate with the broker. The publisher publishes data without any information regarding the identity, location or the number of subscribers. Similarly, the subscribers receive the data without any information regarding the publishers.

Unlike in traditional client-server paradigm where the client cannot send messages to the server when the server is not operational, in publish-subscribe paradigm regardless of publisher or subscriber, each can continue to operate normally. This is referred to as *Time Decoupling*.

Scalability

The other main advantage of this pattern is scalability. The publish-subscribe pattern provides the room for better scalability compared to traditional client-server due to message caching, parallel operation, network-based routing, etc [29]. Even outside the enterprise world publish-subscribe paradigm has proven its by providing a broad range of distributed messages via protocols such as Atom (standard) and Rich Site Summary (RSS).

In the realm of publish-subscribe the subscribers have control on what content they are interested in receiving rather than receiving all the content. This is achieved by a technique called *filtering*, which also led to the different types of publish-subscribe systems [23]. We discuss two of the many types in the following sections.

2.4.1 Topic-based publish-subscribe systems

One of the popular adoptions of the publish-subscribe paradigm is the Topic-based publish-subscribe model. Although there is a subtle difference, it is also referred as subject-based or channel-based filtering. In this approach, there exists a named logical channel called "topics". Publishers publish content/event to a given topic and subscribers interest in receiving the content/event subscribe to the topic to receive them. All the subscribers will receive the same content from a topic to which the publishers publish the content.

It simplifies matching by having a static publisher and subscriber relationship. At the time of publication, the subscriber set is known. For a given topic, every content/event published on a topic is received by the same subscriber set unless topics or subscriptions change. Figure 2.5 depicts a topic-based publish-subscribe model.

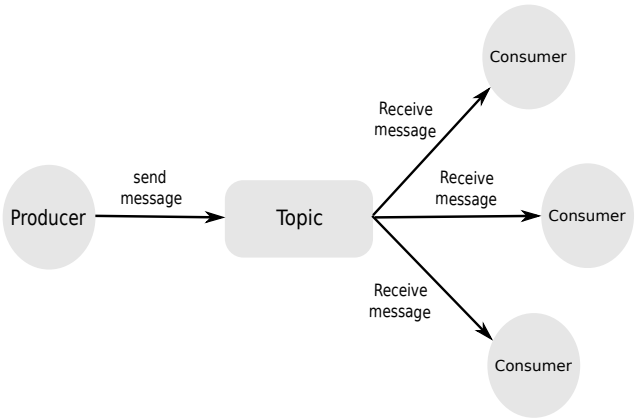


Figure 2.5: Topic-based publish-subscribe

Topics are hierarchically organized. Subscriptions can contain wildcards to match multiple topics; however, the publisher cannot use wildcards within a topic to publish event/content.

Hierarchically organized topics

- E.g., news/sports/cricket

– E.g., news/sports/football

Topic subscription using wildcard

– E.g., news/sports/*

2.4.2 Content-based publish-subscribe systems

Content-based publish-subscribe system provides more flexibility [3] and expressiveness to subscriber in expressing interest [50]. It provides a query based approach to come filters over the content by using a *subscription language*. Thus provides more precise matching of interest. Figure 2.6 shows an example of a content-based publish-subscribe system.

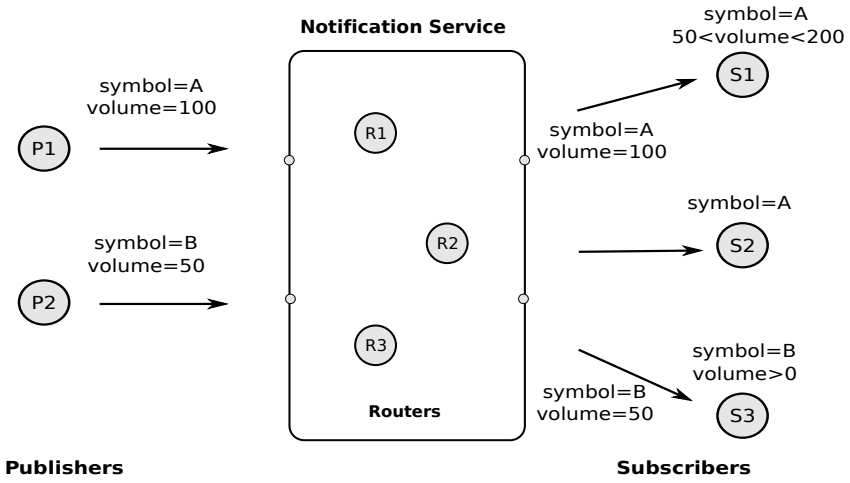


Figure 2.6: Content-based publish-subscribe

Unlike topic-based model, here the notifications of events are grouped based on the runtime query calculations rather than predefined constraints. This provides high flexibility and more expressiveness, but these advantages also comes at a cost. Since the event/content receivers are not known previously, the interested subscribers are determined at runtime requiring more resources.

2.5 Java Message Service

To handle the producer-consumer problem Java provides a Java Message Oriented Middleware API called the Java Message Service (JMS) API for sending and receiving messages between two or more clients [27]. JMS being a part of the Java Enterprise Edition (JEE) allows application based on Java EE to create, send, receive and also read messages. It provides the provision for communication between the various components to be loosely coupled, asynchronous and reliable in a distributed application.

JMS API allows communication that is not just loosely coupled but also the following:

- **Asynchronous:** Messages can be delivered to a client as they arrive by a JMS provider. The client need not request messages to receive them.
- **Reliable:** JMS API provides different levels of reliability for message delivery. It can be that the message needs to be delivered only once or any other lower reliability level in case an application is not concerned about missing or duplicate messages.

2.5.1 JMS Elements

The elements of JMS are described below:

- **JMS Provider:** It is an implementation of the JMS interface either as a Java implementation or an adapter to a non-Java implementation.
- **JMS Client:** Any application that produces and/or receives messages.
- **JMS producer:** Also referred as publisher. Any JMS client that creates and sends messages.
- **JMS consumer:** Also referred as a subscriber. Any JMS client that is interested in receiving the messages.
- **JMS message:** It can be any data that is being transmitted between two or more JMS clients.
- **JMS queue:** It is a staging area where the messages that are waiting to read are contained. The message can be read by only one consumer. Unlike its name *queue*

suggests, the messages need not received in the same order as they were sent. The JMS queue is only responsible for every message to be processed only once.

- **JMS topic:** A logical channel for multiple consumers to receive the same message sent by producers.
- **JMS connection factory:** It is an object that encapsulates administrator defined configuration parameters that can be used to create a connection to a provider.
- **JMS destinations:** It is an object that represents the target for messages that the client produces or the source of messages that client consumes. The destination can either be a queue or a topic or both.
- **JMS administered objects:** The JMS connection factory objects and JMS destination objects are both managed administratively rather than programmatically and hence they are termed as administered objects.
- **JMS sessions:** Sessions are a single threaded context that can be used either for producing and consuming messages. Entities such as message producer, message consumers and also messages are created using the session object.

2.5.2 JMS API Architecture

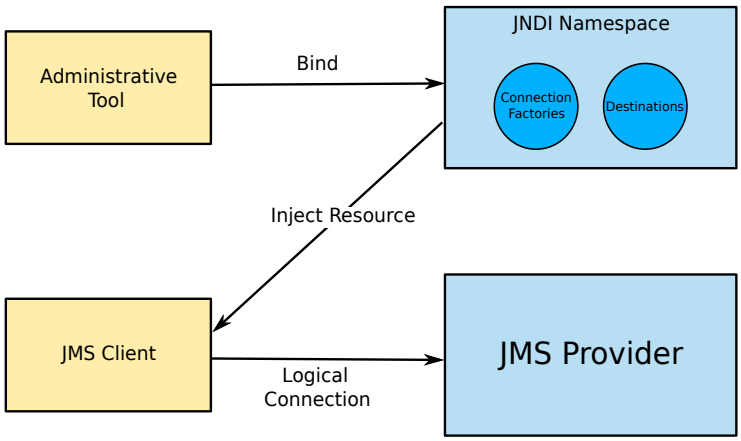


Figure 2.7: JMS API Architecture

Figure 2.7 depicts the way the JMS components interact [35]. The administrative tools do the binding between the connection factories and destinations into a JNDI namespace. The JMS client uses resource injection to access the objects in the namespace and establish a logical connection to the same objects via the JMS provider.

2.5.3 JMS API Programming Model

Figure 2.8 shows how the basic building blocks of JMS interact in a JMS client application. The connection factory creates the connection object with in turn is used to create a session. As discussed before a session object can be used to create a message, message producer, and message consumer.

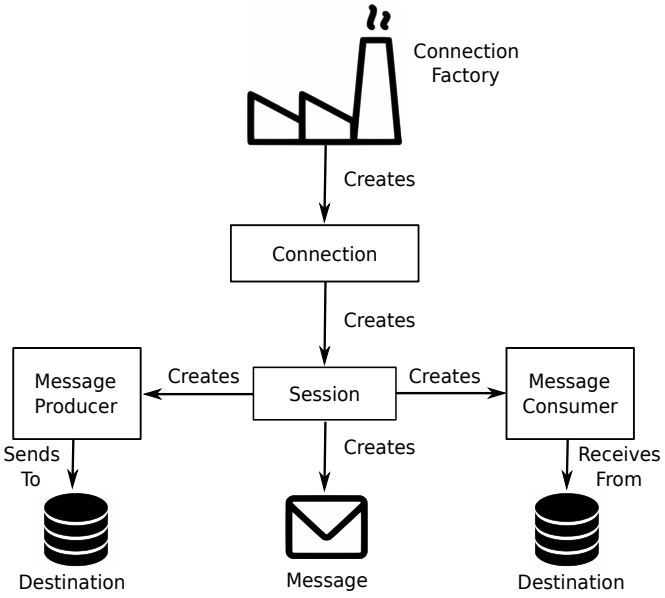


Figure 2.8: JMS API Programming Model

The JMS programming API minimizes the concepts that are required to build messaging systems at the same time facilitates sufficient features to even maintain sophisticated messaging systems.

2.6 MQTT

MQTT is a lightweight messaging protocol [39] and an ISO standard [34]. It is a publish-subscribe based protocol for use on top TCP/IP. For non-TCP/IP networks used by embedded devices there exist a variant of MQTT called *MQTT-SN* typically used in Wireless Sensor Networks (WSN) [33]. It is best suited for connections where the network bandwidth is limited. It was created by Dr. Andy Stanford-Clark and Arlen Nipper in the year 1999 [42]. The latest version of the protocol is V3.1.

2.6.1 MQTT methods

MQTT defines methods to perform actions on selected resources. The representation of the resources depends on the implementation of the server. The resource could be a file or outside from any executable on the server.

Below are the methods defined by MQTT:

- **Connect:** Waits for the MQTT client to establish a connection with the server.
- **Disconnect:** Waits till the MQTT client completes its work and for the session to disconnect.
- **Subscribe:** Waits for the subscription to complete.
- **UnSubscribe:** Asks the server to unsubscribe a client from one or multiple topics.
- **Publish:** Passes the request to the MQTT client.

2.6.2 Security

The latest version of the MQTT protocol supports authentication by passing username and password. It also supports encryption over the network via SSL. However, as SSL is not a very light weight it adds a considerable amount of overhead. One way to avoid this is to encrypt the data thats is transmitted but this is not native to the protocol.

2.7 Related Work

To the best of our knowledge, the only work that has been done in the area of Shared Dicitonary Compression (SDC) within publish-subscribe model is *SDC for Publish-Subscribe (SSPS)* [20] which has the implementation on a centralized broker. Another similar work *Publish/Subscribe for Mobile Applications using Shared Dictionary Compression* [21] is by three of the authors of SSPS. Expanding the scope outside publish-subscribe we find early works like *Shared Dictionary Compression for HTTP (SDCH)* [36] and *Dictionary Compression for a Scan-Based, Main-Memory Database System* [13]. We will discuss these work briefly in this section.

2.7.1 Shared Dictionary Compression for HTTP

Four employees at Google proposed SDCH in the year 2008. It is an HTTP/1.1 compatible extension aimed at reducing required bandwidth via the use of shared dictionary between the client and the server. Currently, browsers like Google Chrome and other Chromium based browsers support SDCH.

The idea was to is to mine the common strings/phrases that occur across contents of multiple pages and also the similarities in the CSS, JavasSript code, etc., in many websites. Once the elements that are common are downloaded, a dictionary is created using those elements. This dictionary is made available to both the browser and the server. Once done, the server can replace the long values using short notations referring the dictionary. On the other end, the browser uses the same dictionary to interpret the short notations and get the long values.

The important considerations are that the browser and the server both must support SDCH and at any instant both must have the same version of the dictionary. If the browser and server refer to different versions of the dictionary, then this would result in a broken page. The conditions for SDCH between the browser and the server are done through GET requests by the browser.

LinkedIn tried the SDCH for their site [49]. For the creation of dictionaries they used the open-source solution *Femtozip* by Garrick Toubassi. They pretend the header of the dictionary for the browser to know the right domain and paths for which the dictionary could be used.

The initial results obtained by LinkedIn demonstrated a remarkable compression ratio

of 81% when SDCH and gzip were combined compared only to gzip for certain files. The result was based on two dictionaries, one for 6225 Javascript files and another one for 1282 CSS files.

2.7.2 Dictionary Compression for a Scan-Based, Main-Memory Database System

This was a master thesis of Janick Bernet at ETH Zurich. The goal was to obeserve the effect of dictionary based compression in a in-memory row-store called Crescando [18] without breaking the predictability and scalability that Crescando provides. It mainly highlights the space savings achieved due to the dictionary based compression.

The dictionary was based on the key-value pairs. A bidirectional data structure called bidi-map (developed as part of the thesis) which provides sub-linear access in both directions was used. The data structure bidi-map was cache-conscious and had low memory footprint.

The result showed a 35% space-savings on segment size of 1Gb on a real world data set from Amadeus. However, there was not significant performance difference with or without compression.

2.7.3 Shared Dictionary Compression in Publish/Subscribe

This was a recent work in June 2016 in which Shared Dictionary Compression was used in the realm of the publish-subscribe model. It is called SDC for Publish-Subscribe (SSPS). It was developed at Informatics department at the Technical University of Munich. The basic idea is to mine the similarities between notifications to build a dictionary that could be shared among the publisher and subscriber. Once both the publisher and subscriber have the dictionary, then they can exchange messages that are compressed using the dictionary.

Just like in SDCH, the open-source solution FemtoZip by Garrick Toubassi is used to create the dictionary. The SDC here is a combination of a dictionary and multiple passes of Huffman Coding. Using this method, the references to the dictionary are represented very efficiently.

The method introduces a new entity called the Sampling Broker (SB) which is respon-

sible for sampling notifications to create dictionaries, maintaining the dictionary and spreading the dictionary to the publishers and subscribers. This entity can either be a part of the broker or can be a separate system.

Unlike in SDCH where the dictionaries are generated upfront, in SSPS the dictionaries are created on the fly when the communication occurs. Initially, the publisher publishes standard uncompressed messages to the broker. The sampling broker gathers these messages and constructs the first dictionary which is then sent to both the publisher and subscriber. Once both the parties have the dictionary, the publisher starts to use the dictionary to compress messages and publishes the compressed message. Since the same dictionary is available with the subscriber, it uses the dictionary to decompress the messages.

The SB has an adaptive algorithm running which observes the compression ratio and bandwidth savings and generates new dictionaries as and when required to maintain the bandwidth savings. We would describe the adaptive algorithm in detail in section 5, as this thesis work is an extension of the SSPS and will be using the same adaptive algorithm.

The dictionaries are also cached so that whenever new subscribers or publishers join, they receive the dictionary. The protocol used for the prototype here was MQTT. In order, to distinguish between the compressed or uncompressed messages the first byte of the MQTT payload contained a code that indicated not only whether the message was compressed or uncompressed but also which version of the dictionary was used by the publisher to compress the message. This is very crucial because a message can be decompressed only using the same dictionary which was used to compress the message.

The result of this work demonstrated an increase in the throughput on limited 2G bandwidth. The dataset used was DEBS15. The experiment evaluation claims a decrease of 40% in time required to send 5000 messages using SDC.

2.7.4 Publish/Subscribe for Mobile Applications using Shared Dictionary Compression

This work is a demonstration of SSPS described previously for mobile applications. The approach of generation and sharing of the dictionary between publisher and subscriber is same as in SSPS.

The protocol and dataset used here is also same as the one in SSPS. The result here demonstrated bandwidth savings up to 88%. The result included the overhead of transmitting the dictionary.

3 Distributed Broker

A distributed broker is a distributed system that aims to tackle the drawbacks of a centralized broker like single point failure, computational bottleneck, etc [19]. As this thesis is about a prototypical implementation of the SSPS on a distributed broker, this chapter discusses the few of the major distributed brokers. This chapter is divided into three section. The first section discusses few of the major distributed brokers. In particular, it discusses the following brokers.

- Apache ActiveMQ
- HornetQ
- Apache Kafka
- RabbitMQ
- Apache ActiveMQ Artemis

The second section discusses different aspects such as scalability, performance, message persistence and so on with regards to these brokers. Finally, the last section explains briefly about our selection of one broker among the discussed brokers for the prototypical SSPS implementation.

3.1 Various Distributed Brokers

3.1.1 Apache ActiveMQ

Apache ActiveMQ is message broker from the Apache Software Foundation that is open source and written in Java. It features a full JMS client via JMS 1.1 [51]. The official site of ActiveMQ claims it to be the most popular and powerful open source

messaging and Integration Patterns system.

ActiveMQ is packed with a huge set of features [6]. It supports various cross-platform clients like Java, PHP, C, C++, C, Python, Ruby and Perl. It supports OpenWire, STOMP, AMQP and MQTT protocols. When it comes to high availability, it supports clustering both via Master Slave and Replicated Message stores using SAN or any shared storage. For persistence, it provides the use of JDBC along with a high-performance journal and Apache ZooKeeper. It provides support for Spring integration and a REST API for language-independent API for messaging. Apart from these, ActiveMQ has many advanced features such as Virtual Destinations and Message Groups.

3.1.2 HornetQ

HornetQ is a high performance, multi-protocol, clustered, embeddable, asynchronous messaging system from JBoss [26]. The work on HornetQ was started by Tim Fox in 2007, and it was released two years later. It is written in Java and provides support for both JMS 1.1 and JMS 2.0 APIs.

Like with most systems, HornetQ also packs many features [30]. In terms of multi-protocol HornetQ supports only STOMP and AMQP. It provides its own core API that can be used instead of JMS. However, the API is only available for Java and not any other language. It provides high availability both using the replication of data store and also via shared file system. It supports clustering to deal with load balancing. For persistence, HornetQ relies on its own high performance journal than relying on slow relational databases. The journal could be either Java NIO or Linux Asynchronous IO (AIO). HornetQ provides REST support in order for clients written in different languages to access natively. HornetQ provides management API to monitor and manage servers [31].

3.1.3 Apache Kafka

Apache Kafka is another open source message broker from the Apache Software Foundation [8]. It is distributed streaming platform written in Scala. Originally Kafka was developed by LinkedIn and was later open sourced in 2011.

Kafka has client in many languages [7] such as Java, C++, Python, etc. Kafka uses a custom binary protocol over TCP rather than using STOMP, AMQP, MQTT or any other

existing protocol [1]. This is mainly because the required control over the protocol in order achieve the desired output that Kafka provides. To support load balancing Kafka supports clustering [25] in which each instance of the broker communicates via ZooKeeper. The cluster state is maintained using ZooKeeper since the brokers are stateless. It provides high availability by via replication in which the followers keep their logs updated with respect to the leader's logs.

Kafka is not just used as a stream processing system but also as a storage system and messaging system. Kafka is used by many enterprises such as Spotify, Netflix and PayPal [17].

3.1.4 RabbitMQ

RabbitMQ is an open source asynchronous messaging broker [48]. It was written in Erlang which is a functional programming language that is well suited for distributed applications. It was developed initially by Rabbit Technologies Ltd. and currently belongs to Pivotal Software, Inc.

RabbitMQ has a rich set of features [46]. It has clients for many languages such as Java, Ruby, PHP, Python and many more. It supports protocols such as AMQP, STOMP, MQTT, and HTTP either directly or via plugins. RabbitMQ supports clustering across LAN and clustering over WAN is not recommended. For high availability, the message queues are replicated to all the slaves. For persistence, RabbitMQ uses its own message store that is shared among all the queues on the server.

RabbitMQ comes with management UI that facilitates the control and monitoring of every aspect of the broker. Also, it also comes with a range of plugins to extend it in different ways, with support to write your own plugin.

3.1.5 Apache ActiveMQ Artemis

Apache ActiveMQ Artemis is another open source messaging system from Apache Software Foundation [5]. It is written in Java. It is an asynchronous, high-performance, multi-protocol, clustered and embeddable system. The current version of ActiveMQ Artemis is 1.4.0.

Apache ActiveMQ Artemis system should not be confused with Apache ActiveMQ.

The code base of HornetQ was donated to the Apache Software Foundation under the codename *Artemis*. Although it is possible that Artemis would become the successor to ActiveMQ eventually, however a final decision is not yet made as per the ActiveMQ board report [32].

Apache ActiveMQ Artemis has support for multiple protocols such as AMQP, MQTT. STOMP and OpenWire. Unlike Apache ActiveMQ, Apache ActiveMQ Artemis provides support for both JMS 1.1 and JMS 2.0. It provides high availability both via replication (non-shared store) and shared store like SAN. It supports clustering for load balancing. For replication, it relies on its own high-performance journal either Java NIO or Java NIO or Linux Asynchronous IO (AIO) rather than any database. It also provides a REST interface for leveraging the scalability and reliability features over a simple interface (REST/HTTP). It provides a management API to manage and interact and can be done over JMS, JMX and core protocol.

From the description and features of Apache ActiveMQ and HornetQ, it can easily be observed that Apache ActiveMQ Artemis packs many useful features from the both the systems. The union between the HornetQ and Apache ActiveMQ community aims to provide a path for the next generation of the message broker.

3.2 Comparison

The comparisons are based on our understanding of these systems from the knowledge gathered from the official documentation and articles over the Internet [9] [22] [53] [44] [40]. As per our knowledge, there exist no direct comparison of all the five systems.

Scalability and Reliability

Since we are looking at distributed brokers here, all the brokers are inherently scalable. All of them support clustering. Similarly, when it comes to reliability, all the brokers are reliable as every one of them support master-slave replication. Table 3.1 lists the ways in which the brokers can handle replication. Data synchronization refers to some form of synchronization between master and slave. It might be using some kind of logs or message store. In shared storage both master and slave share the same data directory using shared file system, the same case is in shared database.

Table 3.1: Replication in brokers

System	Replication
Apache Kafka	Data synchronization
RabbitMQ	Data synchronization
HornetQ	Data synchronization or Shared storage
Apache ActiveMQ Artemis	Data synchronization or Shared Storage
Apache ActiveMQ	Data synchronization or Shared storage or Shared database

Performance

It is tough to get a statistical comparison of all five the systems, the reason being the configuration, features, and different protocols. Almost, every broker praise about their performance. Many claim to be the fastest but this valid only for short span and specific scenario. For example, [47] shows the performance for HornetQ far superior to ActiveMQ. Another article [28] shows Kafka being superior to ActiveMQ and RabbitMQ.

So the selection criteria comes to the purpose and application for which the broker is intended to be used. Kafka is suitable for streaming applications. RabbitMQ is good for advanced messaging pattern involving routing and load balancing. ActiveMQ is good in performing without persistence [38]. ActiveMQ Artemis has its own charm as having combined features of HornetQ and ActiveMQ.

Messaging Protocol

Different brokers support a different set of protocols. Some support standard protocols and some like Apache Kafka have their own custom protocol. Table 3.2 lists the protocols that are supported by different brokers.

3.3 Our Selection

Our selection of the broker for the prototypical implementation of the SSPS on a distributed broker was based on two criteria. One being the language in which the

Table 3.2: Protocol support in brokers

Broker	Protocols
Apache ActiveMQ	OpenWire, STOMP, AMQP, MQTT
HornetQ	STOMP, AMQP
Apache Kafka	Custom binary protocol
RabbitMQ	STOMP, AMQP, MQTT
Apache ActiveMQ Artemis	OpenWire, STOMP, AMQP, MQTT

broker was written and second being the MQTT protocol as SSPS was designed around MQTT.

We decided to choose *Apache ActiveMQ Artemis* as it contained the features of both HornetQ and ActiveMQ. HornetQ is already donated to Apache Software Foundation under the codename Artemis. ActiveMQ Artemis will eventually be replacing ActiveMQ, and thus it makes sense to pick the latest system. We eliminated Apache Kafka because it only supports its custom binary protocol. We ruled out RabbitMQ because it's written in Erlang which is entirely new to us and our language expertise was Java.

4 Apache ActiveMQ Artemis

This section describes the necessary concepts and features of Apache ActiveMQ Artemis that are required to understand for implementing the SSPS model in Apache ActiveMQ Artemis. The chapter contains seven sections. The first section describes the architecture of ActiveMQ Artemis. The second section explains on how to use the server. The third section describes the use of JMS with ActiveMQ Artemis. The fourth section describes the persistence in ActiveMQ Artemis. The fifth section describes how clustering can be achieved on ActiveMQ Artemis. The sixth section describes the high availability and failover aspect of the system. Finally, the seventh section describes interceptors in ActiveMQ Artemis.

4.1 Architecture

Core Architecture

The core of Apache ActiveMQ Artemis is a simple set of Plain Old Java Objects (POJOs). It is designed in a way to use as few external dependencies as possible. The makes Apache ActiveMQ Artemis to be embedded in any project or injected via dependency injection.

Apache ActiveMQ Artemis comes with its own extreme high-performance journal which is used for message and other persistence. This persistent journal allows Apache ActiveMQ Artemis to get high performance that just cannot be obtained by the utilization of a relational database for persistence.

Apache ActiveMQ Artemis currently offers two APIs for the client side that can be used for messaging.

- **Core client API:** An API that allows clients to use complete functionality of messaging without few of the complexities of JMS.

- **JMS client API:** This is a standard JMS API for the client.

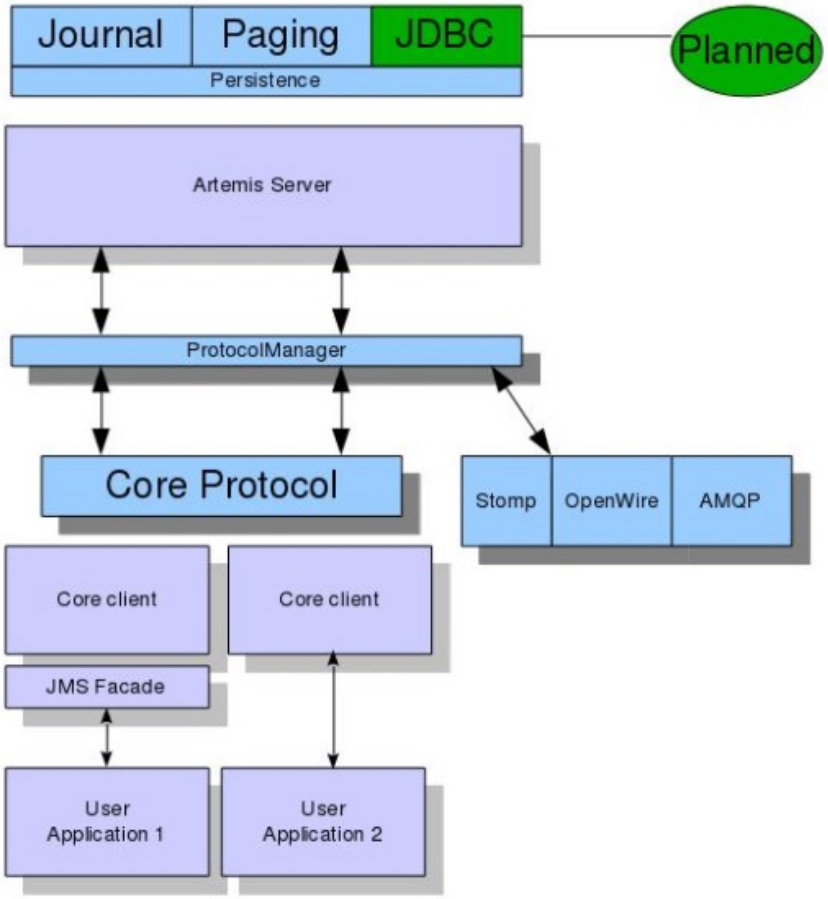


Figure 4.1: Apache ActiveMQ Artemis High Level Architecture

Apache ActiveMQ Artemis supports protocols such as OpenWire, STOMP, AMQP, and MQTT. Apart from these, a client can choose to communicate using JMS as well via the core protocol. Apache ActiveMQ Artemis is protocol agnostic and hence it does not understand JMS. When the client uses the JMS API, the JMS interactions are translated to the core API using the wire format of the Apache ActiveMQ Artemis. On the client side, a JMS facade layer is used to implement JMS semantics.

Figure 4.1 shows the high level architecture of Apache ActiveMQ Artemis. It also shows

the interaction of two applications. One application interacts using the JMS client API and another application uses the core client API for communication with the server. A facade layer can be seen for the application that uses the JMS client API for interaction.

4.2 Using the server

Installation

The source code of Apache ActiveMQ Artemis can be downloaded from its repository [4]. After downloading the project, it can be build using the commands in Listing 4.1.

```
$ mvn -Prelease install
```

Listing 4.1: Building Apache ActiveMQ Artemis

Once the build is complete, the distribution is located under the folder *artemis-distribution*. The distribution contains the relevant folders shown in Listing 4.2.

```
|___ bin
|
|___ web
|
|___ examples
|
|___ lib
|
|___ schema
```

Listing 4.2: Apache ActiveMQ Artemis Folders

- **bin:** Contains scripts and binaries to run the system.
- **web:** Location where the web context is loaded when the system runs.
- **examples:** Contains many example programs.
- **lib:** Contains required libraries and jars.
- **schema:** Contains XML schemas for validating configuration files..

Creating a Broker Instance

To create a broker instance by the name testbroker from the Artemis distribution navigate to the *bin* directory and execute the command in Listing 4.3.

```
$ ./artemis create testbroker
```

Listing 4.3: Creating Broker Instance

The following folders will appear under the created broker directory.

- **bin:** Contains executables related to the broker instance.
- **etc:** Contains configuration files.
- **data:** Contains files used for storage of persistent messages.
- **log:** Contains log files.
- **tmp:** Contains any temporary files.

Main configuration file

Under the directory *etc* a file named *broker.xml* is the main configuration file that is used by the Apache ActiveMQ core server.

Starting and Stopping a Broker Instance

To start the broker instance navigate to the *bin* directory of the broker instance and execute the command in Listing 4.4.

```
$ ./artemis run
```

Listing 4.4: Starting Broker Instance

To stop the broker instance the same *artemis* script is used but with a stop argument. Listing 4.5 is an example.

```
$ ./artemis stop
```

Listing 4.5: Stopping Broker Instance

4.3 Using JMS

In addition to the JMS agnostic API Apache ActiveMQ Artemis provides a JMS API for messaging. Many users are more comfortable using JMS as it is a very popular API messaging standard. In this section, we explain on how to use JMS with Apache ActiveMQ Artemis using a simple example. The example has one JMS queue named *exampleQueue*, a single message producer named *MessageProducer* that sends a message to the queue and a single consumer called *MessageConsumer* consuming the message from the queue.

4.3.1 JNDI Configuration

As per the JMS specification the administered objects such as the JMS queue, topic and connection factory instances has to be made available by the JNDI API. The configurations are specified in a file name *jndi.properties* which needs to be added to the classpath of the project.

ConnectionFactory JNDI

A client uses the connection factory to make connections to the server by knowing the server location and other configuration parameters.

Different connection factories can be used [2]. Listing 4.6 shows an example where an ActiveMQInitialContextFactory connection factory type is used and tcp connection to localhost is provided along with the port number 61616.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.
    ↪ ActiveMQInitialContextFactory
connectionFactory.ConnectionFactory=tcp://localhost:61616
```

Listing 4.6: ConnectionFactory JNDI

Destination JNDI

Just like connection factories JMS destinations can also be configured via JNDI. It is a name-value entry. The pattern for the name to be followed is *topic.<jndi-binding>* or *queue.<jndi-binding>*. The value is the name of the queue.

Listing 4.7 shows an example by extending the Listing 4.6 example. In this example the binding takes place to the queue named *exampleQueue* hosted on the Apache ActiveMQ Artemis server, specified in the broker.xml file as *<queue name="exampleQueue"/>*.

```
java.naming.factory.initial=org.apache.activemq.artemis.jndi.
    ↪ ActiveMQInitialContextFactory
connectionFactory.ConnectionFactory=tcp://localhost:61616
queue.queues/exampleQueue=exampleQueue
```

Listing 4.7: ConnectionFactory JNDI

4.3.2 Example code

Listing 4.6 describes the final code for our example.

```
1
2 //Create a JNDI initial context to lookup JMS objects
3 InitialContext initialContext = new InitialContext();
4
5 //Look up the connection factory to create connections to
    localhost:61616
6 ConnectionFactory connectionFactory = (ConnectionFactory)
    initialContext.lookup("ConnectionFactory");
7
8 //Look up the Queue
9 Queue orderQueue = (Queue)initialContext.lookup("queues/
    exampleQueue");
10
11 //Use the connection factory to crete a JMS connection
12 Connection connection = connectionFactory.createConnection()
    ;
```

```
13
14 //Create a JMS session with auto acknowledge mode.
15 Session session = connection.createSession(false, Session.
       AUTO_ACKNOWLEDGE);
16
17 //Create a message producer to send messages to the queue
18 MessageProducer messageProducer = session.createProducer(
       exampleQueue);
19
20 //Create a message consumer to receive/consume messages
       from the queue
21 MessageConsumer messageConsumer = session.createConsumer(
       exampleQueue);
22
23 //start the connection
24 connection.start();
25
26 //Send a simple text message
27 TextMessage textMessage = session.createTextMessage("This
       is an example message");
28 messageProducer.send(textMessage);
29
30 //Receive or consume the message
31 TextMessage receivedMessage = (TextMessage)messageConsumer.
       receive();
32
33 //Print the message
34 System.out.println("Received message: " + receivedMessage.
       getText());
```

Listing 4.8: Example JMS code

4.4 Persistence

Apache ActiveMQ Artemis comes with a high-performance journal that is used for persistence. This file journal is highly optimized for storage and offers excellent performance.

Journals in Apache ActiveMQ Artemis are append only journals. It is a set of files. Each of the files is pre-created by a fixed size and filled with padding initially. As and when operations are performed such as adding a message, deleting a message, update a message, records are appended in the journal. When a journal is full, the operations are done on the next journal.

Using a sophisticated garbage collection algorithm Apache ActiveMQ Artemis determines when a particular journal is no longer needed. Journals are compacted using the compaction algorithm of Apache ActiveMQ Artemis which removes dead space from the journal.

4.4.1 Journal Implementations

Apache ActiveMQ Artemis has two implementations of the journal.

Java NIO

This implementation offers very good performance and runs on any platform having JRE 6 and above. It uses the Java NIO to interface with the file system.

Linux Asynchronous IO

This implementation typically provides better performance than Java NIO implementation of the journal. It uses a native wrapper to the talk to Linux Asynchronous IO library (AIO). However, this runs only on Linux kernel 2.6 and above.

4.4.2 Journals

All the journals are configured in the *broker.xml* file located in the Apache ActiveMQ broker instance.

Bindings journal

This journal stores bindings information like the set of queues and their attributes that are deployed on Apache ActiveMQ Artemis server. This journal is comparatively slower than message journal. It is always Java NIO journal. These journal files have the file extension as *bindings*.

JMS journal

This journal is only created if JMS is used. It stores all the JMS-related data such as topics, queues and connection factories. Any JMS resources created through the management API is also stored here. These journal files have file extension as *jms* extension.

Message journal

As the name suggests this journal stores the message data and duplicate-id caches. By default, the server tries to use AIO journal and switches to NIO journal if the AIO is not available. These journal files have file extension as *amq* extension.

Large message and Paging

Apache ActiveMQ Artemis persists large messages outside the message journal in a separate directory called *large-messages*. In the case of low memory situations, Apache ActiveMQ Artemis provides an option to enable paging to page messages.

Zero Persistence

In certain scenarios, if required Apache ActiveMQ Artemis can also be configured to use zero persistence. If this is set, then no data such as binding information, message data, large messages and paging will be persisted.

4.5 Acceptors and Connectors

Acceptors and connectors define how an Apache ActiveMQ Artemis server accepts a connection and how clients can connect to an Apache ActiveMQ Artemis server respectively.

Both acceptors and connectors have to be configured in the *broker.xml* file. Listing 4.9 and 4.10 shows example for configuring a acceptor and a connector respectively.

```
<acceptors>
   <acceptor name="netty">tcp://localhost:61616</acceptor>
</acceptors>
```

Listing 4.9: Configuring Acceptor Example

```
<connectors>
   <connector name="netty">tcp://localhost:61616</connector>
</connectors>
```

Listing 4.10: Configuring Connector Example

4.6 Clusters

Clusters allow load sharing among the Apache ActiveMQ Artemis servers. Each node that is active manages its own connections and messages. Clusters can be formed in different topologies such as symmetric cluster and chain cluster.

A cluster can be formed by each node in the cluster configuring cluster connections to other nodes in the main configuration file *broker.xml*. An internal core bridge is created when node forms a cluster with another node. These connections allow the messages to flow among the nodes to balance the load among the nodes in the cluster.

4.6.1 Server discovery

In a cluster, the nodes need to propagate their connection details to other nodes. This is done via server discovery. Server discovery can be done either using UDP multicast or JGroups.

Broadcast Groups

The way in which a server or client can connect to another server is defined by a connector. The means used by a server to broadcast connectors are called broadcast groups. Broadcast groups are defined in the *broker.xml* configuration file. Listing 4.11 shows an example of broadcast group by the name *bg-group1*. The group address and the port form the multicast address to which the data will be broadcast. The period between successive broadcasts is set 1000 milliseconds and the connector used is netty-connector [43].

```
<broadcast-groups>
   <broadcast-group name="bg-group1">
      <group-address>${udp-address:231.7.7.7}</group-address>
      <group-port>9876</group-port>
      <broadcast-period>1000</broadcast-period>
      <connector-ref>netty-connector</connector-ref>
   </broadcast-group>
</broadcast-groups>
```

Listing 4.11: Broadcast Group Example

Discovery Groups

The way in which the broadcast endpoint receives the connector information is defined by discovery groups. A discovery group keeps a list of entries of connectors for each server from which it receives the broadcast information. This is updated as and when new broadcast information is received. A server entry is deleted if the broadcast information from that server is not received before the timeout period. Similar to broadcast groups, discovery groups are also defined in the *broker.xml* configuration file. Listing 4.12 shows an example of a discovery group by the name *dg-group1*. The group address and the port form the multicast address to listen on. A refresh timeout of 5000 milliseconds is set. This indicates the period a discovery group waits after the previous broadcast from a server before removing the server from its list of entries.

```
<discovery-groups>
   <discovery-group name="dg-group1">
      <group-address>${udp-address:231.7.7.7}</group-address>
      <group-port>9876</group-port>
```

```
        <refresh-timeout>5000</refresh-timeout>
    </discovery-group>
</discovery-groups>
```

Listing 4.12: Discovery Group Example

4.6.2 Cluster connection configuration

For load balancing between the nodes of a cluster, cluster configuration has to be defined in the *broker.xml* file. Listing 4.13 shows an example of a cluster configuration. A cluster connection with name my-cluster is defined for the address that matches the string "jms". The connector used is netty-connector and the Listing 4.12 discovery group is used.

```
<cluster-connections>
    <cluster-connection name="my-cluster">
        <address>jms</address>
        <connector-ref>netty-connector</connector-ref>
        <discovery-group-ref discovery-group-name="dg-group1"/>
    </cluster-connection>
</cluster-connections>
```

Listing 4.13: cluster connection configuration Example

4.7 High Availability and Failover

The ability of a system to continue operating after one or more failure of the servers is referred to as high availability. Failover being a part of high availability is the ability of the client applications to continue operating by migrating from one server to another when a server fails.

4.7.1 Live - Backup Groups

Apache ActiveMQ Artemis provides failure by allowing servers to be linked as live-backup pairs. A live server can have many backup servers. When the live server fails,

one of the backup server takes over the live server. Apache ActiveMQ Artemis provides two different ways to achieve successful failover; one is data replication, and the other is shared storage.

4.7.2 Data Replication

When replication is used, the live and the backup servers do not share the same persistent directories such as paging, large messages, bindings and journal directories. All the synchronization happens over the network during normal operations. Thus, the data on the live server is replicated on the backup servers.

The synchronization of data between the live and backup servers happens in parallel to the normal operation, and thus it does not cause blocking on connected clients. When the live server fails, the backup server needs to synchronize all the data from the live server before it takes over the live server. The time for synchronization depends on the amount of data and the connection speed.

The configuration for using replication is defined in the *broker.xml* file. Listing 4.14 represents the configuration of shared storage for the master server. Listing 4.15 represents the configuration of shared storage for backup servers.

```
<ha-policy>
   <replication>
      <master/>
   </replication>
</ha-policy>
```

Listing 4.14: HA replication master

```
<ha-policy>
   <replication>
      <slave/>
   </replication>
</ha-policy>
```

Listing 4.15: HA replication slave

4.7.3 Shared Storage

When shared storage is used, both the live and the backup servers (slaves) share the same data directories such as paging, large messages, bindings and journal directories using a shared file system. When a live server fails, the backup server takes over by loading the required files from the shared file system. The clients can then connect to it. Typically some high-performance Storage Area Network (SAN) is used.

One advantage of using shared storage is that there is no overhead of replication during the normal operation. However, this also becomes a disadvantage as the backup server needs to load all the data from the shared storage.

The configuration for using shared storage is defined in the *broker.xml* file. Listing 4.16 represents the configuration of shared storage for the master server. Listing 4.17 represents the configuration of shared storage for backup servers.

```
<ha-policy>
   <shared-store>
      <master/>
   </shared-store>
</ha-policy>
```

Listing 4.16: HA Shared storage master

```
<ha-policy>
   <shared-store>
      <slave/>
   </shared-store>
</ha-policy>
```

Listing 4.17: HA Shared storage slave

4.8 Intercepting Operations

To intercept packets coming into the server or going out of the server, Apache ActiveMQ Artemis provides *interceptors*. Interceptors allow the packets to be modified or inspected. Interceptors are dependent on the protocol for, e.g., to intercept MQTT packets MQTTInterceptor interface has to be used by the classes that need to intercept the packets.

Interceptors are of two types, incoming interceptors, and outgoing interceptors. Both interceptors are configured in the *broker.xml* file.

Incoming Interceptor

An incoming interceptor intercepts packets entering the server. Listing 4.18 shows example configuration of an incoming interceptor implemented by the class *PropertyInterceptor*.

```
<remoting-incoming-interceptors>
   <class-name>org.apache.activemq.artemis.core.protocol.mqtt.
      ↪ PropertyInterceptor</class-name>
```

```
</remoting-incoming-interceptors>
```

Listing 4.18: Configuring Incoming Interceptor Example

Outgoing Interceptor

An outgoing interceptor intercepts packets exiting the server. Listing 4.19 shows example configuration of an outgoing interceptor implemented by the class *ConnectionInterceptor*.

```
<remoting-outgoing-interceptors>
   <class-name>org.apache.activemq.artemis.core.protocol.mqtt.
      ↪ ConnectionInterceptor</class-name>
</remoting-outgoing-interceptors>
```

Listing 4.19: Configuring Outgoing Interceptor Example

List of Figures

List of Tables

Bibliography

[1] *A Guide To The Kafka Protocol*. https://cwiki.apache.org/confluence/display/KAFKA/A+Guide+To+The+Kafka+Protocol. Accessed: 2016-05-26.

[2] *Administering JMS Connection Factories and Destinations*. https://docs.oracle.com/cd/E18930_01/html/821-2416/ablkb.html. Accessed: 2016-07-19.

[3] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. "Matching Events in a Content-based Subscription System." In: *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '99. Atlanta, Georgia, USA: ACM, 1999, pp. 53–61. ISBN: 1-58113-099-6. DOI: 10.1145/301308.301326.

[4] *Apache ActiveMQ Artemis Repository*. https://github.com/apache/activemq-artemis. Accessed: 2016-06-02.

[5] *Apache ActiveMQ Artemis Website*. https://activemq.apache.org/artemis/. Accessed: 2016-05-27.

[6] *Apache ActiveMQ Website*. http://activemq.apache.org/. Accessed: 2016-05-26.

[7] *Apache Kafka Clients*. https://cwiki.apache.org/confluence/display/KAFKA/Clients. Accessed: 2016-05-26.

[8] *Apache Kafka Website*. https://kafka.apache.org/intro.html. Accessed: 2016-05-26.

[9] T. Bayer. *ActiveMQ, Qpid, HornetQ and RabbitMQ in Comparison*. https://www.predic8.com/activemq-hornetq-rabbitmq-apollo-qpid-comparison.htm. Accessed: 2016-05-27.

[10] T. Bell, I. H. Witten, and J. G. Cleary. "Modeling for Text Compression." In: *ACM Comput. Surv.* 21.4 (Dec. 1989), pp. 557–591. ISSN: 0360-0300. DOI: 10.1145/76894.76896.

[11] J. Bentley and D. McIlroy. "Data compression using long common strings." In: *Data Compression Conference, 1999. Proceedings. DCC '99*. Mar. 1999, pp. 287–295. DOI: 10.1109/DCC.1999.755678.

[12] J. Bentley and D. McIlroy. "Data Compression with Long Repeated Strings." In: *Inf. Sci.* 135.1-2 (June 2001), pp. 1–11. ISSN: 0020-0255. DOI: 10.1016/S0020-0255(01)00097-4.

[13] Bernet, Janick. *Dictionary Compression for a Scan-Based, Main-Memory Database System*. http://e-collection.library.ethz.ch/eserv/eth:1184/eth-1184-01.pdf. Accessed: 2016-10-18.

[14] P. A. Bernstein. "Middleware: A Model for Distributed System Services." In: *Commun. ACM* 39.2 (Feb. 1996), pp. 86–98. ISSN: 0001-0782. DOI: 10.1145/230798.230809.

[15] K. Birman and T. Joseph. "Exploiting Virtual Synchrony in Distributed Systems." In: *SIGOPS Oper. Syst. Rev.* 21.5 (Nov. 1987), pp. 123–138. ISSN: 0163-5980. DOI: 10.1145/37499.37515.

[16] A. K. Y. Cheung and H.-A. Jacobsen. "Load Balancing Content-Based Publish/-Subscribe Systems." In: *ACM Trans. Comput. Syst.* 28.4 (Dec. 2010), 9:1–9:55. ISSN: 0734-2071. DOI: 10.1145/1880018.1880020.

[17] *Companies using Kafka*. https://cwiki.apache.org/confluence/display/KAFKA/Powered+By. Accessed: 2016-05-26.

[18] *Crescando*. https://www.ecc.ethz.ch/research/crescando. Accessed: 2016-10-18.

[19] *Distributed Broker*. http://zeromq.org/whitepapers:brokerless. Accessed: 2016-10-19.

[20] C. Doblander, T. Ghinaiya, K. Zhang, and H.-A. Jacobsen. "Shared Dictionary Compression in Publish/Subscribe Systems." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 117–124. ISBN: 978-1-4503-4021-2. DOI: 10.1145/2933267.2933308.

[21] C. Doblander, K. Zhang, and H.-A. Jacobsen. "Publish/Subscribe for Mobile Applications using Shared Dictionary Compression." In: *ICDCS Demos*. 2016.

[22] J. Eriksson. *Comparing message-oriented middleware for financial assets trading*. 2016.

[23] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. "The Many Faces of Publish/Subscribe." In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078.

[24] *FemtoZip*. https://github.com/gtoubassi/femtozip. Accessed: 2016-04-30.

[25] N. Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[26] P. Giacomelli. *HornetQ messaging developer's guide*. Packt Publishing Ltd, 2012.

[27] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. "Java message service." In: *Sun Microsystems Inc., Santa Clara, CA* (2002).

[28] M. Heo. *Kafka vs. RabbitMQ vs. ActiveMQ*. `http://mungeol-heo.blogspot.de/2015/01/kafka-vs-rabbitmq-vs-activemq.html`. Accessed: 2016-05-27.

[29] M. HiveMQ Enterprise. *Broker,"MQTT Essentials Part2: Publish & Subscribe"*.

[30] *HornetQ Features*. `https://developer.jboss.org/wiki/HornetQFeatures`. Accessed: 2016-05-26.

[31] *HornetQ Website*. `http://hornetq.jboss.org/`. Accessed: 2016-05-26.

[32] *How does ActiveMQ compare to Artemis*. `http://activemq.apache.org/how-does-activemq-compare-to-artemis.html`. Accessed: 2016-05-27.

[33] U. Hunkeler, H. L. Truong, and A. Stanford-Clark. "MQTT-S x2014; A publish/-subscribe protocol for Wireless Sensor Networks." In: *Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on*. Jan. 2008, pp. 791–798. DOI: 10.1109/COMSWA.2008.4554519.

[34] *Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1*. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=69466`. Accessed: 2016-05-25.

[35] *JMS API Architecture*. `http://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html`. Accessed: 2016-10-16.

[36] Jon Butler, Wei-Hsin Lee, Bryan McQuade, Kenneth Mixter. *A Proposal for Shared Dictionary Compression over HTTP*. `http://lists.w3.org/Archives/Public/ietf-http-wg/2008JulSep/att-0441/Shared_Dictionary_Compression_over_HTTP.pdf`. 2008.

[37] T. J. Kruk. "Prescript to the lectures Distributed Systems (DS)(work in progress)." In: (2006).

[38] G. Kuntal. *Message Queue Comparision*. `http://kuntalganguly.blogspot.de/2014/08/message-queue-comparision.html`. Accessed: 2016-05-28.

[39] D. Locke. "Mq telemetry transport (mqtt) v3. 1 protocol specification." In: *IBM developerWorks Technical Library], available at http://www. ibm. com/developerworks/webservices/library/ws-mqtt/index. html* (2010).

[40] H. Mike. *Message Queue Shootout!* `http://mikehadlow.blogspot.de/2011/04/message-queue-shootout.html`. Accessed: 2016-05-27.

[41] *Moquette*. `https://projects.eclipse.org/proposals/moquette-mqtt`. Accessed: 2016-04-30.

[42] *MQTT FAQ*. http://mqtt.org/faq. Accessed: 2016-10-17.

[43] *Netty Project*. http://netty.io/. Accessed: 2016-07-21.

[44] Z. Peter. *Exploring message brokers*. https://dzone.com/articles/exploring-message-brokers. Accessed: 2016-05-27.

[45] P. R. Pietzuch and J. Bacon. "Hermes: A Distributed Event-Based Middleware Architecture." In: *Proceedings of the 22Nd International Conference on Distributed Computing Systems*. ICDCSW '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 611–618. ISBN: 0-7695-1588-6.

[46] *RabbitMQ Features*. https://www.rabbitmq.com/features.html. Accessed: 2016-05-27.

[47] *RabbitMQ Performance Measurements, part 1*. https://www.rabbitmq.com/blog/2012/04/17/rabbitmq-performance-measurements-part-1/. Accessed: 2016-05-27.

[48] A. Richardson et al. "Introduction to RabbitMQ." In: *Google UK, Sep* 25 (2008).

[49] O. Shapira. *Shared Dictionary Compression for HTTP at LinkedIn*. https://engineering.linkedin.com/shared-dictionary-compression-http-linkedin. Accessed: 2016-04-30.

[50] H. Shen. "Content-Based Publish/Subscribe Systems." In: *Handbook of Peer-to-Peer Networking*. Ed. by X. Shen, H. Yu, J. Buford, and M. Akon. Boston, MA: Springer US, 2010, pp. 1333–1366. ISBN: 978-0-387-09751-0. DOI: 10.1007/978-0-387-09751-0_49.

[51] B. Snyder, D. Bosanac, and R. Davies. "Introduction to Apache ActiveMQ." In: *Active MQ in Action* (), pp. 6–16.

[52] Spronk, René. *The role of Message Brokers*. http://www.ringholm.de/docs/00100_en.htm. Accessed: 2016-10-14.

[53] Y. Trudeau. *Exploring message brokers*. https://www.percona.com/blog/2014/05/05/exploring-message-brokers/. Accessed: 2016-05-27.

[54] *Types of Middleware*. https://apprenda.com/library/architecture/types-of-middleware/. Accessed: 2016-10-13.