



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/Subscribe 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

Acknowledgments	iii
Abstract	iv
1 Introduction	1
1.1 Shared Dictionary Compression	2
1.2 Problem	2
1.3 Goals and Contribution	3
1.4 Thesis Outline	4
2 Background	6
2.1 Middleware	6
2.2 Message Brokers	8
2.3 Clusters	9
2.3.1 Cluster Attributes	9
2.4 Publish-subscribe systems	11
2.4.1 Topic-based publish-subscribe systems	12
2.4.2 Content-based publish-subscribe systems	13
2.5 Java Message Service	14
2.5.1 JMS Elements	14
2.5.2 JMS API Architecture	15
2.5.3 JMS API Programming Model	16
2.6 MQTT	17
2.6.1 MQTT methods	17
2.6.2 Security	17
2.7 Related Work	18
2.7.1 Shared Dictionary Compression for HTTP	18
2.7.2 Dictionary Compression for a Scan-Based, Main-Memory Database System	19
2.7.3 Shared Dictionary Compression in Publish/Subscribe	19

Contents

2.7.4 Publish/Subscribe for Mobile Applications using Shared Dictionary Compression	20
List of Figures	22
List of Tables	23
Bibliography	24

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 [6].

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 [9]. 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[3] 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. McIlroy in [4] and subsequently extended in [5].

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 [12] is an open-source solution by Garrick Toubassi [25].

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 [9]. 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 [20]. 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

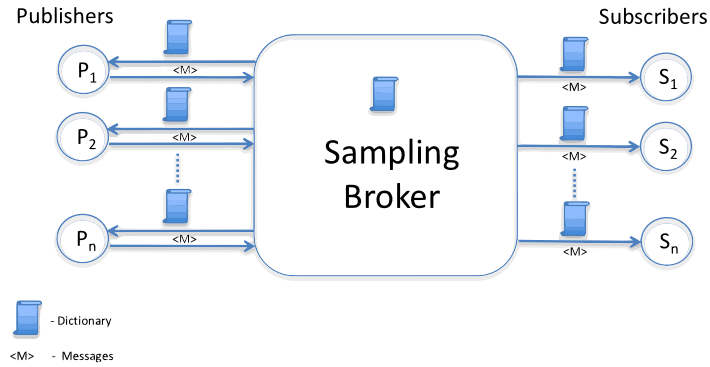


Figure 1.1: Sampling Broker in SSPS model

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.

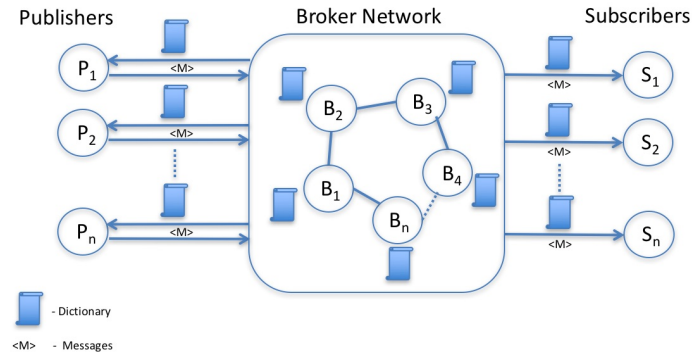


Figure 1.2: SSPS in a distributed broker

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 [19]. 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.

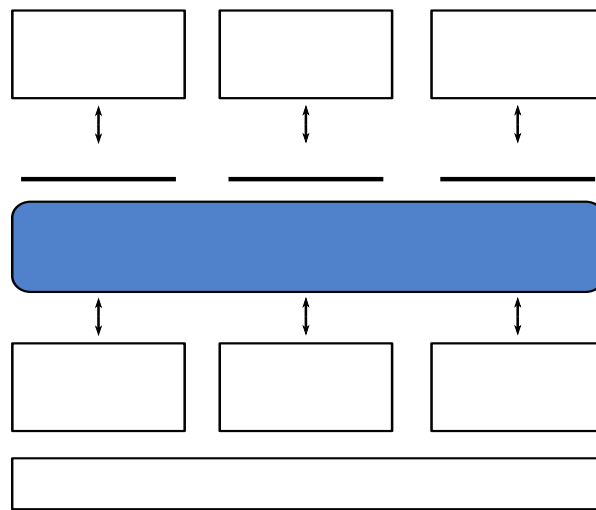


Figure 2.1: Middleware

There are many types of middleware [**middleware_types**]. 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 [18]. 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.

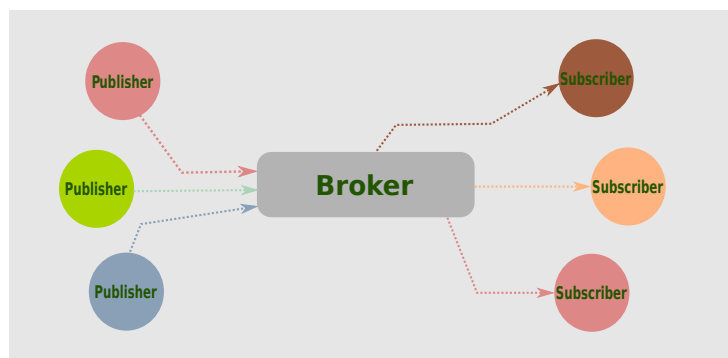


Figure 2.2: Message Broker

While the primary function of a message broker is to decouple a sender and receiver, it can also perform other activities [24] 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

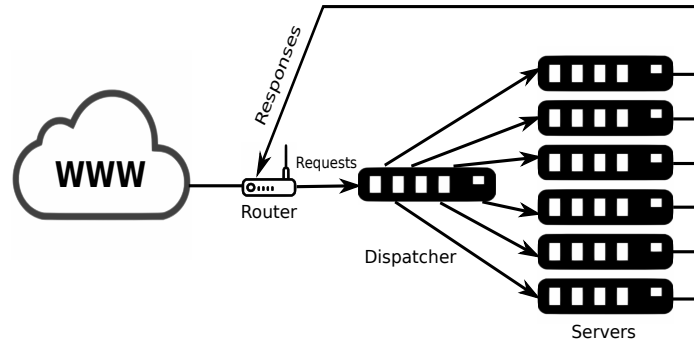


Figure 2.3: Load-balancing cluster Example

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.

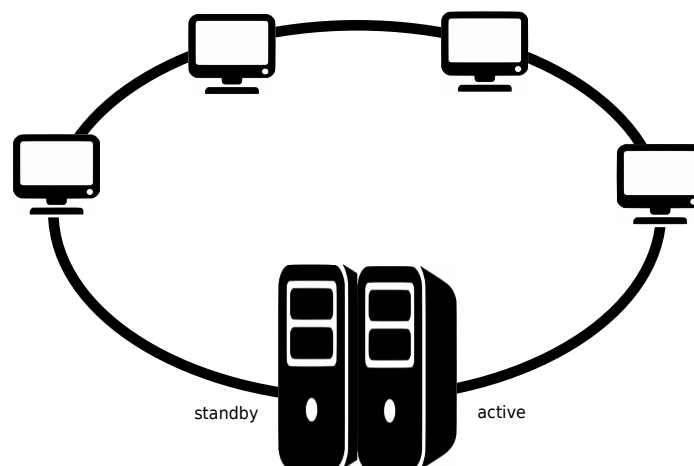


Figure 2.4: High-availability cluster Example

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 [23].

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 [7]. 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. 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) [1].

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 [11]. 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.

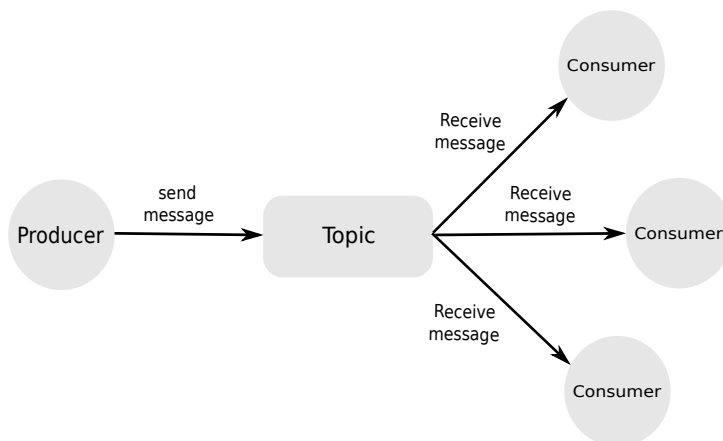


Figure 2.5: Topic-based publish-subscribe

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 [2] and expressiveness to subscriber in expressing interest [26]. 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.

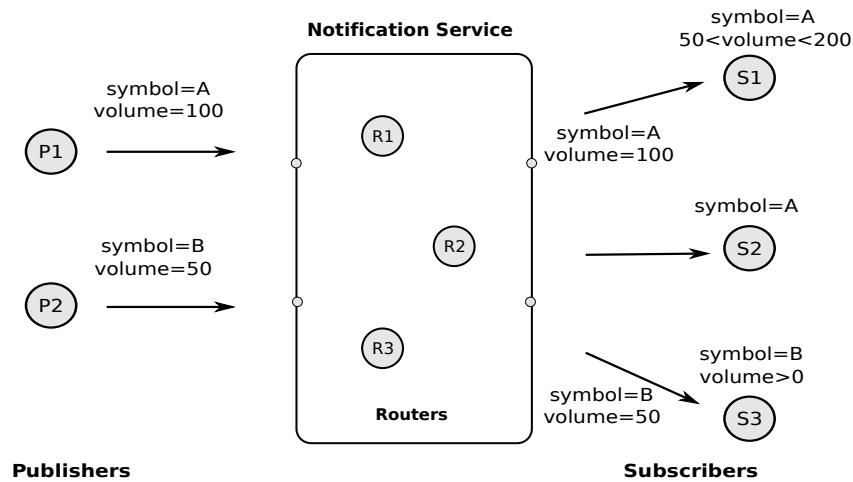


Figure 2.6: Content-based publish-subscribe

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 [15]. 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*

Figure 2.7 depicts the way the JMS components interact [16]. 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.

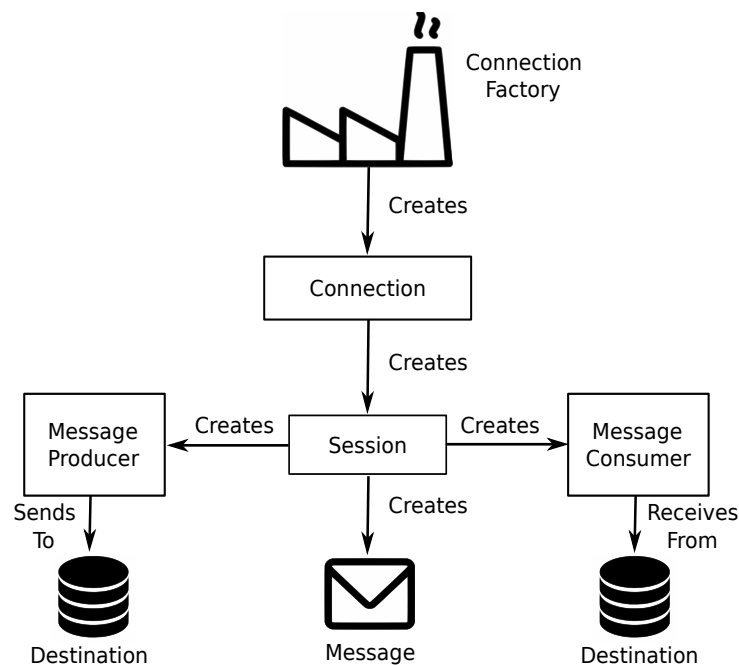


Figure 2.8: JMS API Programming Model

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 and an ISO standard [22]. 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) [13]. 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 [21]. 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 that 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 Dictionary Compression (SDC) within publish-subscribe model is *SDC for Publish-Subscribe (SSPS)* [9] which has the implementation on a centralized broker. Another similar work *Publish/Subscribe for Mobile Applications using Shared Dictionary Compression* [10] 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)* [17] and *Dictionary Compression for a Scan-Based, Main-Memory Database System* [14]. 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, JavaScript 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 [25]. 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 observe the effect of dictionary based compression in a in-memory row-store called Crescendo [8] without breaking the predictability and scalability that Crescendo 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.

List of Figures

1.1	Sampling Broker in SSPS model	3
1.2	SSPS in a distributed broker	4
2.1	Middleware	7
2.2	Message Broker	8
2.3	Load-balancing cluster Example	10
2.4	High-availability cluster Example	10
2.5	Topic-based publish-subscribe	12
2.6	Content-based publish-subscribe	13
2.7	JMS API Architecture	15
2.8	JMS API Programming Model	16

List of Tables

2.1 Clusters vs. Distributed systems	9
--	---

Bibliography

- [1] *Advantages of publish-subscribe pattern*. https://en.wikipedia.org/wiki/Publish-subscribe_pattern. Accessed: 2016-10-15.
- [2] 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.
- [3] 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.
- [4] 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.
- [5] 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.
- [6] 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.
- [7] 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.
- [8] *Crescendo*. <https://www.ecc.ethz.ch/research/crescendo>. Accessed: 2016-10-18.
- [9] 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.
- [10] C. Doblander, K. Zhang, and H.-A. Jacobsen. “Publish/Subscribe for Mobile Applications using Shared Dictionary Compression.” In: *ICDCS Demos*. 2016.
- [11] 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.
- [12] *FemtoZip*. <https://github.com/gtoubassi/femtozip>. Accessed: 2016-04-30.
- [13] 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.
- [14] Janick Bernet. *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.
- [15] *Java Message Service API*. https://en.wikipedia.org/wiki/Java_Message_Service. Accessed: 2016-10-16.
- [16] *JMS API Architecture*. <http://docs.oracle.com/javase/6/tutorial/doc/bncdx.html>. Accessed: 2016-10-16.
- [17] Jon Butler, Wei-Hsin Lee, Bryan McQuade, Kenneth Mixer. *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.
- [18] T. J. Kruk. “Prescript to the lectures Distributed Systems (DS)(work in progress).” In: (2006).
- [19] *Middleware Wikipedia*. https://en.wikipedia.org/wiki/Middleware#cite_ref-MW-00_1-0. Accessed: 2016-10-13.
- [20] *Moquette*. <https://projects.eclipse.org/proposals/moquette-mqtt>. Accessed: 2016-04-30.
- [21] *MQTT FAQ*. <http://mqtt.org/faq>. Accessed: 2016-10-17.
- [22] *MQTT Protocol*. <https://en.wikipedia.org/wiki/MQTT>. Accessed: 2016-10-17.
- [23] 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.

- [24] *Purpose of message broker*. https://en.wikipedia.org/wiki/Message_broker. Accessed: 2016-10-14.
- [25] *SDC LinkedIn*. <https://engineering.linkedin.com/shared-dictionary-compression-http-linkedin>. Accessed: 2016-04-30.
- [26] 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.