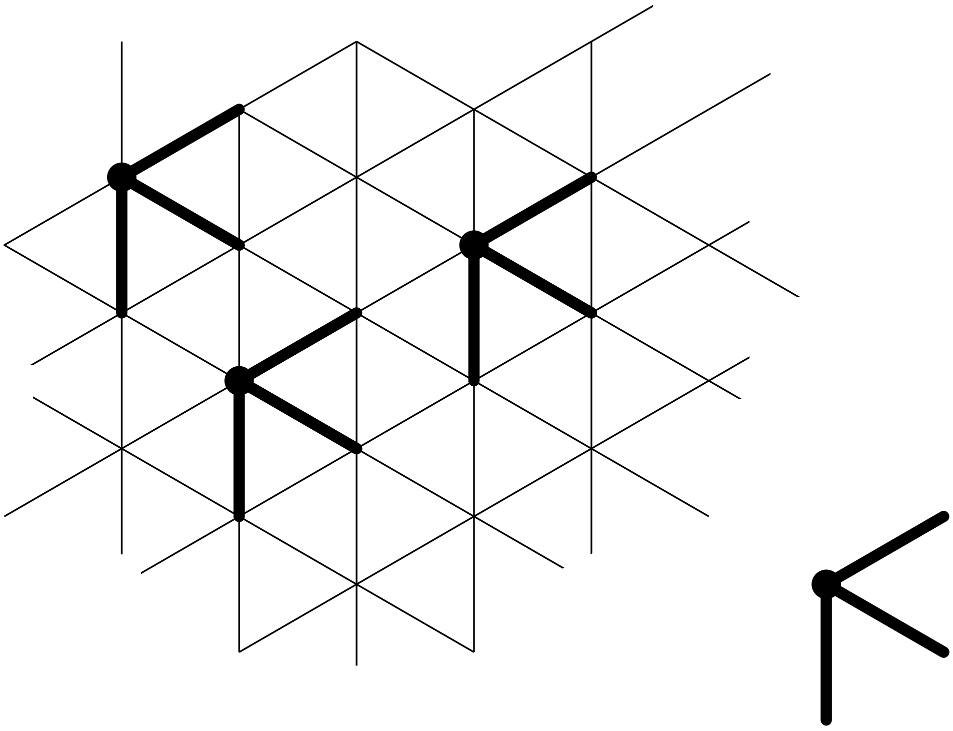
CHALMERS
UNIVERSITY OF TECHNOLOGY



A Distributed Publish/Subscribe System built on a DHT Substrate

Master of Science Thesis in the Program Computer Systems and Networks

André Laszlo

Department of Computer Science & Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

A Distributed Publish/Subscribe System built on a DHT Substrate

André Laszlo

©André Laszlo, January 2016.

Examiner: Olaf Landsiedel

Cover: Vector graphics by the Author, 2014

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden January 2016

Acknowledgments

I would like to thank Andrius Kašėta, Kristin Alehammar, Martin Sylvesten and Per-Åke Minborg at Chilirec for providing the inspiration and setting for this thesis.

My managers at HiQ and Ericsson have been helpful and understanding during my work. Thank you Ulrika Muchow Stenström, Simona Bamerlind, Andreas Sandberg and Torbjörn Lundin, as well as Andreas Lindfalk, formerly at HiQ, who provided technical coaching initially. I am also grateful towards Andrej Lamov who helped me with proofreading and comments.

I would also like to take the opportunity to express my gratitude towards my examiner Olaf Landsiedel, for his excellent feedback and for pushing me in the right direction when I needed it.

Finally, a special thanks is in order to Narjis Hachilif for her constant support and limitless patience, as well as for proofreading and feedback.

André Laszlo, Gothenburg, January 2016

Abstract

The publish/subscribe pattern is commonly found in messaging systems and message-oriented middleware. When large numbers of processes are publishing messages in applications where low latency and high throughput is needed, the performance of the messaging system is critical. Several solutions exist that provide high throughput and low latency to a high number of concurrent processes, such as RabbitMQ and Kafka.

What happens to the performance of the system when each process also has a complex or large set of subscriptions? This is the case when users of an internet radio application notify each other of songs currently being played and the subscriptions of each user correspond to the user's wish list – a list of songs that the user is interested in recording.

This thesis examines how the popular messaging systems RabbitMQ and Kafka handle this situation when topic-based message filtering is used to model subscriptions. A prototype messaging system, Ingeborg, which is built on the key-value store Riak is also introduced and its performance is examined and compared to the previously mentioned systems.

The results of the experimental study show that it is difficult to scale both RabbitMQ and Kafka with regards to the number of topics used, but that RabbitMQ shows greater flexibility in its configuration. Finally, the prototype system Ingeborg shows that it is possible to design a messaging system from scratch based on a key-value store, allowing even greater flexibility in prioritizing trade-offs and performance properties of a system.

Contents

1

Introduction

The year 2014 marked an important milestone for the music industry. For the first time, revenues from digital sales were as big as revenues from the sales of CDs and other physical formats. Almost a quarter of digital revenues, reaching a total of 6.85 billion USD, came from music subscription services. These services had an estimated 41 million users, 13 million more than the year before.[1]

Meanwhile, music piracy, or the act of making unauthorized copies of music, is seen as an enormous problem by the recording industry and other copyright holders world-wide. The Recording Industry Association of America, RIAA, claims that billions of songs are being illegally downloaded and that only a minority of all music acquired by U.S. consumers has been paid for.[2]. The International Federation of the Phonographic Industry, IFPI, writes in their annual *Digital Music Report* that 20% of all fixed line internet users are using services that offer copyright infringing music and that 4 billion music downloads are being made through the BitTorrent protocol.[1]

In Sweden, recording songs played on the radio is not considered copyright infringement, as long as the recordings are only made for personal use. [3, §12, §§26 k-m] Artists are remunerated through fees collected by national collective rights management organizations for music copyright holders. There are several such organizations with distinct responsibilities, such as

Stim, Sami and CopySwede. Anyone who plays copyrighted music in public may be obligated to pay licensing fees to Sami, representing artists, or to Stim, another organization that acts on behalf of composers and other copyright holders. The organization CopySwede collects fees covering loss of income due to copies made for personal use. The fees are currently covering a wide range of devices, such as cassette tapes, mobile phones with digital storage, writable CDs, and hard drives. The fee for an external hard drive, for example, is one Swedish krona (SEK) per GB for devices smaller than 80 GB and 80 SEK for larger hard drives. For smartphones the fee is 3.5 SEK per GB, with no upper limit.[4]

Chilirec[1] was an application, developed by a company from Gothenburg with the same name. The idea behind the application was that people should be able to find, record and organize music from all the thousands of online radio stations available today. Music would be free for the users, since they would record songs off the radio for personal use. Artists would at the same time receive money from the radio stations through the collective rights management organizations.

The first version of Chilirec was launched in 2007 and had a simple interface where the user could start recording immediately after opening the site in their web browser. The actual recording was done on the server, and several stations were recorded at once. Songs were stored in the user's online music library. The powerful server downloading music concurrently from multiple online radio stations would quickly fill the music library with songs. The landing page of Chilirec's web site at the time told visitors "After one day you have more than 50 000 songs". [5]

The music industry questioned the company's interpretation of Swedish law, arguing that the process was too automatic to constitute "copying for personal use". The online storage was also questioned, since the files would be stored on Chilirec's servers and not on the user's

[1]`http://www.chilirec.com`

computer[2]. As a response to pressure from the music industry, the web recorder was shut down and development of a desktop version began.

The desktop application was launched in mid-September 2008. To reduce data traffic and to eliminate most of the criticized automatization, a wish list concept was introduced. The user would add songs to the wish list and the server would listen to hundreds of stations at once, alerting the desktop client whenever a song on the wishlist was being played. The application would immediately connect to the station playing the song and start recording it.

After the launch in the fall of 2008, Chilirec gained thousands of users world-wide. User growth was good initially but eventually slowed down, probably due to competition from streaming services[3] and music available through BitTorrent. This led the founders of Chilirec to focus on other projects, while the service was left running. User growth and revenue from the paid premium version were both small and in 2012 the idea and the remains of the application were handed over to a team at Chalmers School of Entrepreneurship, a combined master's program and technology pre-incubator.

The new team started researching the market and got to work developing new business models, prototypes and services, and also a new version of the application addressing some of the technical limitations found in previous versions. The new application would be faster, easier to use and have a new design, but most importantly it would not rely on a centralized server for wishlist alerts. Instead, each client would listen to a few stations each and notify other clients about new songs playing, using a peer-to-peer notification system. This would reduce operational costs dramatically, since the server would go from constantly listening to hundreds of radio stations, to not listening to any stations at all. Costs related to bandwidth

[2]This interpretation might sound strict today, but this took place a couple of years before "cloud storage" became the norm. Dropbox was founded in 2007, for example, and did not officially launch until 2008.[6]

[3]Spotify, currently the biggest online music streaming service, was launched the month after the desktop version of Chilirec.[7]

usage would be reduced dramatically, as well as the general server load in terms of processing and storage.

The goal of this thesis was to create and evaluate a prototype that solves this problem. Two clients in the system should be able to inform each other whenever online radio stations start playing new songs. Each client has a "wish list", a list of songs that they want to record. A client does not have knowledge of the other clients in the system or of their wish lists. Instead, when the client is listening to a station and a new song starts playing, it will send a notification to the system about this event. Other clients that have this song in their wish lists will then, very quickly, be notified of the event so that they can start recording the song.

1.1 Motivation

Publish/subscribe systems are "solving a vital problem pertaining to timely information dissemination and event delivery from publishers to subscribers".[8, p. 1] Thanks to the usefulness and flexibility of this design pattern, publish-subscribe (pub/sub) systems are used in a wide range of applications. They are used in GUIs for coupling the different interface widgets to each other; in push systems where real-time content is sent to users; in targeted delivery and information filtering systems; in signaling planes to ensure real-time delivery of messages between components; in Service Oriented Architecture (SOA); in Complex Event Processing (CEP), for example in algorithmic trading; in cloud computing; in Internet of Things applications; and in online multiplayer games.[8, pp. 26-27]

Distributed Hash Tables (DHT) (see section 2.2) are sometimes used as a substrate for distributed pub/sub systems. Scribe [9] is one example of such a system; it is a pub/sub system built on top of the DHT Pastry [10]. A problem with these architectures is that reliability, latency and throughput is largely dependent on the nodes that make up the overlay network, for example when broadcasting to the nodes in a multicast tree. Reliable delivery, in this case, is hard to guarantee. Even more so for latency guarantees.

DHT clients are complex since they need to maintain the overlay network and to perform other book-keeping. This also means that look-up speeds can be unpredictable. One solution could be to simplify producers and consumers and let them connect to a "centralized DHT" – a distributed system of nodes participating in a DHT which connecting clients perceive as a single service.

Pub/sub systems fall broadly into two categories, message queues with pub/sub routing semantics, such as RabbitMQ; and peer-to-peer systems where the pub/sub mechanisms are sometimes part of the protocol, such as in the Willow[11] protocol, and sometimes built on top of it as in the Scribe[9] system. The first kind often has a maximum practical number of topics or subscriptions and the second kind often has a combination of high latency and low throughput.

While these systems are both very successful and useful, there are still many potential use-cases for systems that promise unlimited topics and subscriptions as well as high throughput and low latency.

1.2 Problem statement

Real-time meta data updates from tens of thousands of online radio stations should be distributed to hundreds of thousands of clients in the system so that they can find the music they are interested in. This thesis will focus on ways to solve the following issues:

- *Decentralization.* Monitoring tens of thousands of streaming servers in a centralized system is expensive, mainly in terms of bandwidth. Therefore, an efficient way of decentralizing the real-time metadata monitoring is needed.

- *Low latency.* When a client connects to a streaming server, the stream will start in the current position in the currently playing song. This means that, when a song starts playing, clients need to receive a notification about the currently playing song and connect very quickly in order not to miss any stream data.

- *Message filtering.* Together, the stations will generate a large amount of metadata, so the metadata information needs to be filtered before it reaches the client. Otherwise, the client will be overloaded with useless metadata and the system will be busy distributing this metadata.

- *Scalability.* When the number of users grows, it should be possible to scale the system horizontally by adding more nodes without increasing latency or other overhead too much. See section 1.3.1.

The above problems taken together outlines a distributed system where data needs to be collected, filtered and distributed quickly. The solution presented by this thesis is built on Riak, a form of DHT and is discussed further in section 2.2 and chapter 4. The hypothesis of this thesis is that Riak will work well for the following purposes:

- As the basis for a distributed routing table, maintaining a big list of all the client subscriptions.

- As a means of looking up a client when a message is to be sent to it.

- As a way to offer synchronization mechanisms for other parts of the system, such as filtering duplicated messages.

The system should handle at least 100 000 concurrent users listening to 20 000 stations. In practice, this means that the system should be able to handle at least 500 000 outgoing messages and 125 000 incoming messages per minute. If these messages are sent as uncompressed

JSON, over the WebSockets protocol, each message will be approximately 200 bytes and the incoming traffic will be approximately 3 Mibit/s and the outgoing traffic 13 Mibit/s. The total throughput, therefore, will be less than 20 Mibit/s. Section 1.3.2 will go into more details regarding throughput.

The latency requirements, further discussed in section 1.3.3, can be defined in terms of percentiles and the number of clients that receives a message within a time frame. The following latency requirements should hold:

- At least 50% of all published events should be delivered to subscribers within 0.5 seconds, to make sure that the average latency is low enough.

- At least 99% of all published events should be delivered to subscribers within two seconds, to give the vast majority of clients a lowest common quality of service.

1.3 Objective

In this section, the objectives of the thesis relating to scalability, throughput and latency will be discussed in further detail.

1.3.1 Scalability

One of the problems (section 1.2) is to achieve horizontal scalability: it should be possible to keep adding nodes if the demand on the system grows. To find out how much the system can grow, the point of maximum capacity needs to be found. In this case capacity is measured as throughput.

1.3.2 Throughput

The smallest acceptable value for the maximum throughput of the system depends on several factors. Assuming that the heaviest load that the system should be able to handle is defined by the following metrics:

- 100 000 concurrent users
- 20 000 broadcasting stations
- The average user listens to five stations
- The average song is 4 minutes long
- The average song is on 100 users' wish lists

The users will report all new events that they see, so they will generate 125 000 incoming messages per minute. Of these events only about 4% are unique since there is a big overlap in the stations that users are listening to, assuming a uniform distribution of listeners over the stations. 100 users are subscribing to the topic for each song, on average, so the outgoing message rate will be 500 000 messages per minute. Since the incoming and outgoing message rates differ in this system, it is necessary to test both in order to understand which one of them is the worst bottleneck.

Measured as a data rate, 625 000 messages per minute will add up to 16.7 megabits/s, assuming that the average message size is 200 bytes and that messages are sent uncompressed. Presumably, bandwidth will not be a bottleneck, since the system's bandwidth will probably be in the order of hundreds of megabits/s or even gigabits/s.

The goal is for the maximum theoretical throughput to be at least 2 100 messages per second for incoming traffic and 8 300 messages per second for outgoing traffic.

1.3.3 Event latency

In this system, the usefulness of a received event degrades significantly as time passes. The metadata of a station changes every couple of minutes, on average, but it is only useful for a second or two since a client that connects to the stream any later than that will not be able to hear the beginning of a song. It is therefore necessary to minimize the processing time of a message passing through the system.

This section will discuss how long a message is valuable and where it is delayed on the way to its destination. Starting with latency, figure 1.1 describes the flow of information about a track change. This information flow can be broken down into the following events, labeled t_1 to t_8:

t_1 – The broadcaster (B) changes its currently playing track.

t_2 – B notifies listeners about the track change by sending an update through its metadata stream.

t_3 – The listening client (C_l) receives the new metadata and processes it.

t_4 – C_l sends information about the track change to the system.

t_5 – The system receives the information about the track change and starts processing it.

t_6 – A client that is subscribing to this type of event (C_s) is found and the system sends a notification to it

t_7 – C_s receives the event and requests the stream from B

t_8 – B receives the request from C_s

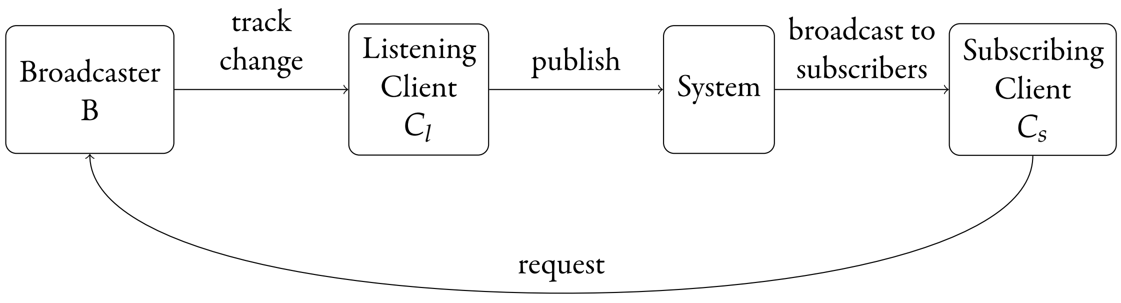


Figure 1.1: The flow of information between broadcasters and clients in the system

The intervals $t_1 - t_3$ and $t_4 - t_5$ depend largely on network latency, therefore most of the design effort should be focused on keeping the interval $t_5 - t_6$, the internal processing time, acceptably small. The metadata processing time, $t_3 - t_4$, should also be taken into account.

What is a reasonable time for $t_1 - t_8$, the full time of the system and client to react and for the server to respond? Servers that broadcast using the common Icecast protocol usually keep a send buffer that contains around at least half a second of audio data. This means that if this interval is smaller than the send buffer, the full song will actually be recorded in its entirety. This would be the best case. The worst acceptable case, from a user perspective, is to not lose more than one or two seconds of the beginning of a track. This is largely dependent on how the track is mixed and other characteristics of the track, but in general a truncation of one second should not be too noticeable.

The first goal is therefore to have at least 99% of all the metadata change events arrive at their destination in less than two seconds. This will ensure that the system delivers a good service to most users.

The second goal is, similarly, to have at least 50% of the events mentioned above arrive within 0.5 seconds. This will enhance the perceived quality of the service, since at least half of the songs will not be noticeably truncated.

1.3.4 Message deduplication

One required feature of this system is message deduplication. This means that if two producers send messages with identical payload, only one of the messages should be delivered. In practice, this means that if two clients are listening to the same station and they both report a track change, a subscriber of that song will only get one event notification.

This concept is related to, but different from, the *at-most-once* semantics of messages queues in Advanced Message Queuing Protocol (AMQP), see section 2.5. At-most-once means that a *specific* message is guaranteed to only be delivered once. In this case, the concept is expanded to include messages with the same payload, sent from *different* producers.

Since both Kafka and RabbitMQ lack native support for this, some system has to be devised to perform deduplication. This will affect the results of the benchmarks, so their results can not be generalized outside of this project. It is unfortunate but necessary since this application requires duplicate messages to be dropped.

A generic solution is needed, which acts like a layer between producers and consumers and filters duplicates. A message that is sent to RabbitMQ or Kafka should be put on a load-balancing queue and consumed by a deduplicator. Since the deduplicators need to share knowledge about messages that have passed through the system recently, some kind of shared, fast cache should be used. For the cache, a system like Memcached[4] or Redis[5] may be used, preferably combined with a node-local cache for better speed. It is worth noting that Redis also has a pub/sub implementation, that will not be used in this project.

[4]`http://memcached.org/`
[5]`http://redis.io/`

1.4 Outline

The next section, Background, will go through some of the background needed to understand modern pub/sub systems in general. It will also go into some details regarding DHTs, message brokers and other technologies used in the Implementation section.

The Related work section goes further into detail on the inner workings of a number of pub/sub systems and previous benchmarks.

An overview of the system is given in the Design section. The protocols and interfaces used are also defined.

The Implementation section outlines how the systems were built. The routing algorithm will also be described, as well as how Riak is used to synchronize the endpoints in the system.

The results of the observations made during the benchmarks are presented in the Evaluation section, as well as how the test platform was implemented, and how it gathered its metrics.

Finally, in the Conclusion section, the results from the evaluation are discussed and put into perspective. Also discussed in this section is whether the goals were met and where, and where there is need for future work.

2

Background

2.1 Publish-subscribe

The general idea of the system described in chapter 4 and chapter 5 is that the system design will use a classical pub/sub pattern. Pub/sub systems, at a generic level, consist of producers that generate messages, and consumers that subscribe to topics that they are interested in. A published message is associated with a topic and will eventually be delivered to some or all of the consumers that are subscribing to the topic. This pattern can be implemented by message-oriented middleware like the one pictured in figure 2.1.

Two common filtering mechanisms in pub/sub systems are *named channels* and *content-based* filtering. When using named channels, each message is published to a specific topic and in content-based filtering, the messages are filtered by the client according to their contents or other attributes.[12]

Messages will be published very frequently and subscriptions will be largely static in the system described in this thesis. This means that if the content filtering would be done on the client side, the clients would spend a lot of resources filtering messages. Therefore, most filtering should ideally be performed before messages reach a client. With the named channel approach, the server keeps a record of which type of message each client is interested in. This

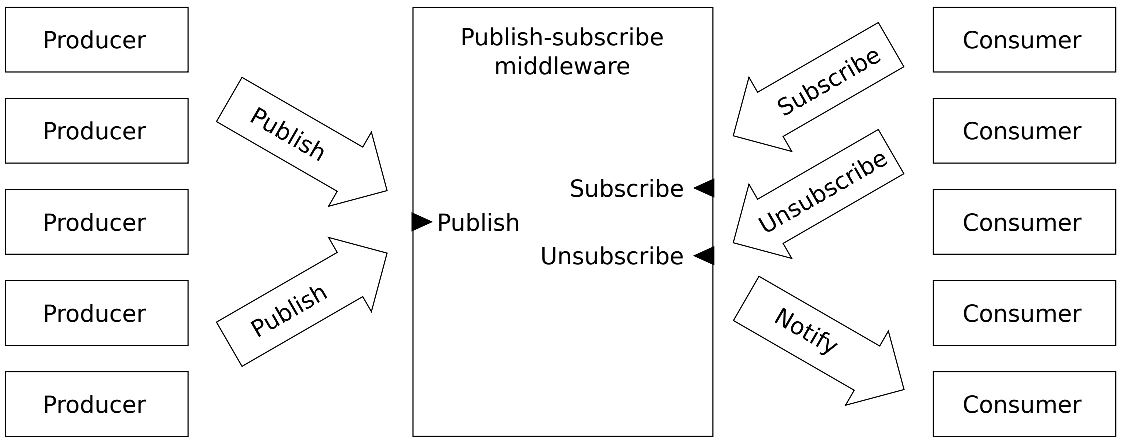


Figure 2.1: A generalized pubsub system.

would be a more efficient option since the channel acts like a filter. Named channels are a little bit less flexible than content-based filtering since support for things like wildcard matching is usually limited or inefficient.

One requirement is that the system should support a very large range of filters, one for every possible song in a wish list, so viewing the system as a set of named channels would result in a number of channels equal to the number of songs known to the clients. However, most of the channels would rarely see any messages. This makes quickly filtering messages and sending them through the right channels more challenging than in a system with fewer channels.

2.2 Distributed hash tables

In the early 2000's, peer-to-peer system researchers were looking for efficient ways to locate data stored on a node in a peer-to-peer network. They were motivated by the realization that the centralized lookup service was a single point of failure in current peer-to-peer applications, such as Napster. The original DHTs: CAN [13], Chord [14], Pastry [10] and Tapestry [15], all published in 2001, solved this problem in similar ways. Later DHTs, such as the ones discussed below, are built on the foundation that these protocols created. All these systems reduced the problem above to the problem of mapping "keys" to "values" and provided a way to do this efficiently, robustly on an Internet-scale – meaning millions of nodes joining and departing from the network frequently.

It might be worth pointing out that the creator of CAN, Sylvia Ratnasamy, predicted that DHTs would be the substrate of "Internet-scale facilities such as global file systems, application-layer multicast, event notification, and chat services"[13], since the system described in this thesis is an instance of using a DHT as a substrate for application-layer multicast.

DHTs evolved rapidly the following years. Kademlia [16], published in 2002, simplified routing by introducing a XOR-based metric that defines a distance between two points in the DHT key space as $d(x, y) = x \oplus y$ where x and y are DHT keys or node IDs, represented as integers. This metric is similar to the one used in Pastry, but is used to build a more efficient network topology. The XOR metric also helped the authors to formally verify parts of the Kademlia protocol. Kademlia has been widely implemented; the DHT currently used by BitTorrent is based on Kademlia [17].

Kademlia has four basic operations: `PING`, `STORE`, `FIND_NODE` and `FIND_VALUE`. All communication in the protocol uses these primitives. Having a small number of operations simplifies the protocol and makes it easier to verify, but it also sets some limitations to what you can achieve using it. Multicast, for example, is not possible in Kademlia. PastryStrings [18] extends Pastry with a distributed data structured called string trees, that allows for rich queries on strings, such as prefix and suffix matching. It also allows a PastryStrings node to generate and subscribe to events.

LightPS [19] and CAPS [20] are two systems that use a hash function called bit mapping (BM) to build a content-based pub/sub on top of a DHT. A subscription in a content-based system often specifies a range of interesting values for each dimension of an interesting event. BM reduces the dimensions of an event to a single dimension in such a way that the range of interesting values can be expressed deterministically by the subscriber as a range of keys. The systems also use a rendezvous mechanism, which in combination with BM allows multiple sources for both subscriptions and events.

Willow [11] is a protocol published in 2005 that uses DHT routing much like Kademlia. It introduces new support for both content- and topic-based pub/sub. Willow's memory requirements and latency grows with $\mathcal{O}(\log n)$ with regards to the number of nodes. Routing is done through logical groups of peers called peer domains. Each Willow node tries to maintain one connection to a peer in each peer domain. At regular intervals, a random peer in each peer group is selected and the previous peer is disconnected if the new latency is lower, which makes the protocol location-aware since shorter hops are preferred over longer ones.

Scribe [9] is a multicast infrastructure that is built on top of the early DHT Pastry [10]. A Scribe node can create, join, leave and send messages to groups. Messages sent to a group will be delivered, with a best-effort guarantee, to all of the group's members. A group membership in Scribe is therefore similar to a subscription to a topic in other pub/sub systems. The delivery of a message to a group is accomplished using a multicast tree, where the message is first delivered to the root node, called the rendezvous node, which delivers it to its children, which are called forwarder nodes, that deliver it to their children and so on.

2.3 Dynamo and Riak

Engineers at Amazon have described a DHT-like system called *Dynamo*[21], which has performed well in several of Amazon's production environments. Since Dynamo is proprietary software, it is necessary to find a licensable or open-source implementation. Riak, released in 2009 and now in wide use, is an open source implementation of the ideas in the Dynamo paper.

Riak, like most of the DHT:s described above, uses a simple key-value storage model and is designed to provide availability, fault-tolerance, and scalability. To avoid key collisions, each key-value pair in Riak goes into a bucket, which acts like a flat namespace. A key, which is a simple binary object, is guaranteed to be unique in each bucket. An object in Riak uses vector clocks for versioning and has a key, a value which is also a binary object and some meta data. The two main interfaces to Riak are its REST and Protocol Buffers APIs[22].

2.4 Erlang/OTP

Erlang is a concurrent programming language designed for programming large-scale, distributed, soft real-time applications.[23] The language was developed at Ericsson, originally for telecom applications, from 1985 to 1998 when it became an open source project.[24] The problems that the language was designed to solve– such as managing high levels of parallelism and programming distributed systems (see table 2.1) – makes Erlang a good tool for building distributed applications.

Erlang is still used by Ericsson in some products, such as the Serving GPRS Support Node (SGSN) [25], and outside Ericsson in big open source projects such as the AMQP implementation RabbitMQ, described in detail in section 3.1, as well as the web server library MochiWeb and the XMPP[1] server Ejabberd.[26]

Erlang has several interesting features that are not commonly found in other languages. Most data structures are immutable by default, which suits the message-passing concurrency paradigm embraced by the language. Message passing is made easy by special operators for sending messages between processes and a rich syntax for pattern-matching any kind of data, such as messages received from other processes. Erlang processes are very cheap in terms of memory and CPU usage, since the Erlang virtual machine has its own scheduler.

[1]Extensible Messaging and Presence Protocol, or XMPP, a communications protocol with several applications such as instant messaging

1	Continuous operation for many years
2	Handling a very large number of concurrent (parallel) activities
3	Interaction with hardware
4	Actions to be performed at a certain point of time or within a certain time
5	Software maintenance (reconfiguration, etc) without stopping the system
6	Stringent quality and reliability requirements
7	Fault tolerance both to hardware failures and software errors
8	Very large software systems
9	Systems distributed over several computers
10	Complex functionality such as feature management.

Table 2.1: Problems that Erlang was designed to solve [27]

Erlang's standard library together with the distributed database Mnesia forms the Open Telecom Platform (OTP). This platform also defines a set of design patterns, called behaviors, that act like interfaces and help programmers create reusable components and modules that interact well with other OTP modules. Two examples of common behaviors are the `application` behavior, which is a module that can be started and stopped as a single unit and be re-used by other applications; and the `gen_server` behavior, that specifies how a process can serve requests from multiple clients. Custom behaviors can also be defined by the programmer or by third-party applications.

When it comes to third-party code for Erlang, there is a rich set of modules, applications and frameworks. The rest of this section will explore some applications relevant to this thesis.

Lager[2] is a logging framework created by Basho, the same company that created Riak (see section 2.3). The framework allows several handlers to be defined for different logging levels. Different logging modules can be used as well, such as console output or file system storage.

[2]`https://github.com/basho/lager`

Cowboy is another HTTP server with WebSocket support. The server was designed with low latency and modularity in mind. An application that needs to communicate over HTTP can do so by creating a module that implements a few specific callbacks.

Other convenient libraries are Jiffy[3] and RiakC[4]. Jiffy is a JSON parser implemented as a Native Implemented Function (NIF), a way to write performance-critical parts of Erlang applications in a language such as C or C++.. RiakC is the official Riak client for Erlang.

2.5 Message brokers

A message broker receives messages from *producers*, processes them and routes them to *consumers*. The clients in this system act both as producers, when publishing information about track changes; and as consumers, when receiving information about tracks in the wish list.

AMQP is an open standard for message oriented systems. The protocol consists of several parts, such as a wire level protocol and a type system. It also defines how queuing, routing, security and reliability should work. Both point-to-point and pub/sub routing is supported.

There are many implementations of AMQP. Some systems include support for AMQP as well as other protocols. RabbitMQ is an open source message broker built on Erlang that, at least partially, implements AMQP.

Two message brokers are discussed into further detail below, RabbitMQ in section 3.1 and Kafka in section 3.2.

2.6 Data Distribution Service

The Object Management Group (OMG) has released a standard for pub/sub middleware called Data Distribution Service for Real-Time System (DDS). It is a standardized model for data-

[3]`https://github.com/davisp/jiffy`
[4]`https://github.com/basho/riak-erlang-client`

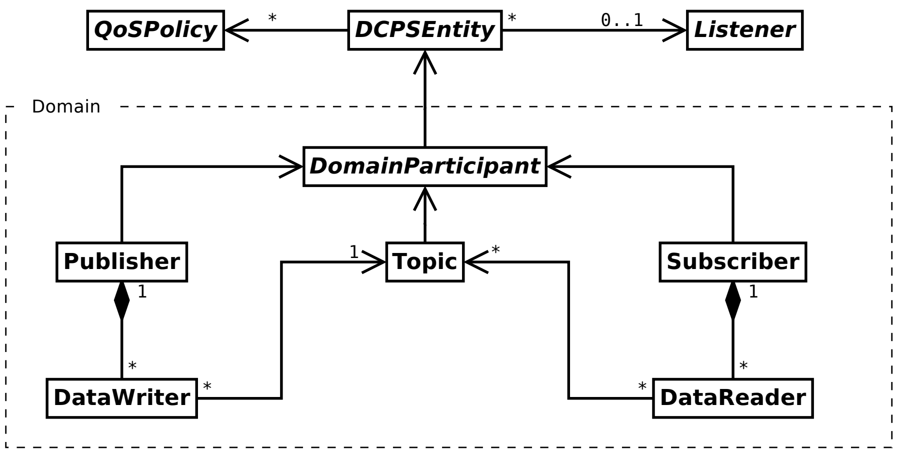


Figure 2.2: The DCPS model of DDS

centric pub/sub (DCPS), designed to enable efficient distribution of data in a distributed system. [28]

The DDS model implements DCPS through the entities in figure 2.2. All entities, or `DomainParticipants`, form a `Domain`, which is a conceptual grouping of the members in a system. Entities can only communicate within a domain. The `Publisher` manages the `DataWriters`, which in turn publishes messages or data to a typed `Topic`. The `Subscriber` manages the `DataReaders`, in the same way. All these entities are subclasses of the `DCPSEntity` base class, which has an attached `QoSPolicy`, and the ability to be notified of events through `Listener` objects.

Each topic defined in DDS is associated with a unique name, a data type and Quality of Service (QoS) information. The type information marks a clear difference between DDS and other systems like AMQP, that are not typed. The DDS type system, called DDS-XTypes is similar to the type system in C and allows a topic to be associated with a primitive type like integer, float, bool, byte or character; or with different kinds of combinations of the primitive types, for example arrays, structs and enumerations.[29]

The typing, QoS, and the fact that each process needs to specify what kind of data it wants to produce and consume in advance, makes it possible to optimize the implementation of DDS while keeping a predictable behavior.

The QoS policies can take several different forms. There is, for example, a deadline policy that DataReaders and DataWriters can use as a contract, saying that the reader expects an update, and the writer commits to produce one at least once during every fixed interval. Another one is the latency budget, allowing a writer to collect and send information in batches, as long as they arrive within a certain acceptable delay to the reader. Interestingly, content-based filtering is also seen as a QoS policy in DDS.

The content-based filter is one of the filtering mechanisms built into DDS, the other two are `Topics` in combination with a `key`, which is used to group data within a topic.

The performance and predictability of DDS makes it suitable for real-time applications. However, the DDS has a significant overhead[28, p. 3] for topic propagation when a large number of topics is used, so DDS-based systems do not scale well with respect to the number of topics used.

2.7 WebSockets

HTTP is a request-response protocol: a client makes a request and the server responds. In HTTP 1.1, the connection between the client and the server is allowed to be kept open between requests to remove the overhead associated with establishing a new connection for each request. Still, HTTP only allows the server to send data to the client as a response to a request.

It is often the case, however, that events emanating from the server side have to be communicated to the client somehow. Several solutions exist. The client can poll the server for updates by sending regular requests, but this will waste requests when there are no updates and the delivery of new events is only as fast as the time between requests. A technique called long

```
GET ws://127.0.0.1:19001/ HTTP/1.1
Host: 127.0.0.1:19001
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: null
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML
    , like Gecko) Chrome/42.0.2311.90 Safari/537.36
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8,sv;q=0.6
Sec-WebSocket-Key: fGA/GZ304AGt/qlACNiItA==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

Code Listing 2.1: WebSocket client handshake message

```
HTTP/1.1 101 Switching Protocols
connection: Upgrade
upgrade: websocket
sec-websocket-accept: h0lPB0mXl553xj/5ZhpE30x3AY8=
```

Code Listing 2.2: WebSocket server handshake response

polling is a little bit more efficient. During long polling, the server delays the response to the client's polling request until it has data to send. There is still some communication overhead for the client in sending a request just to allow the server to respond. Additional overhead is introduced by the separate connections needed for incoming and outgoing messages. There are also issues with network timeouts during the server delay.[30]

The WebSocket protocol, created by the IETF's *BiDirectional or Server-Initiated HTTP* working group, is a solution to all of the above problems. It is designed to allow bidirectional communication within the existing HTTP infrastructure. The protocol consists of an initial handshake, followed by the actual data transfer. A WebSocket handshake is initiated by the client, which sends a regular HTTP request, such as the one seen in code listing 2.1. If the server implements WebSockets and the `Upgrade` field has the value `websocket`, it will respond with the the HTTP code 101, "Switching Protocols", seen in code listing 2.2.[31]

After a successful handshake, the client and server will both use the existing connection for communication using WebSocket messages, consisting of one or more *frames*. There are three types of frames: UTF-8 encoded text frames, binary data frames, and control frames. The control frames are used for protocol-level signaling, for example to let the other end know that the connection is closing, or for pinging and ping responses ("pong") used for keep-alive purposes or to verify the responsiveness of the other end.

Since both the original HTTP connection and the WebSocket connection assumes a reliable underlying transport protocol the protocol is robust with regards to message loss, while having lower latency than previous solutions. It also has message-ordering guarantees, messages will arrive at the other end in the same order that they were sent.

2.8 Benchmarking pub/sub systems

It is important to verify that a system's performance meets requirements. While theoretical models such as time and memory complexity analysis can be useful, the performance characteristics of a real system, with its hardware, software and network equipment can often be more easily assessed using benchmarks.

There are many standardized benchmarks for different types of systems and applications, but hereinafter the word will be used in the most generic sense – "to evaluate [a system's] performance under a particular workload for a fixed configuration"[32] – and not necessarily referring to a standardized test.

A benchmark should be relevant, fair, repeatable, verifiable and economical. Relevant can mean a lot of different things depending on the context and scope of the benchmark, but it should at least be stressing hardware and software in a way that is similar to the way they are used in a production environment, and present measurements that are meaningful, understandable and representative.[33]

3

Related work

Some of the DHTs discussed in section 2.2 lack production-ready implementations. Others are implemented as parts of applications or as libraries. Kademlia, for example, is used in the Mainline BitTorrent client through the Python library Khashmir.

Message Oriented Middleware (MOM) can be defined as "any middleware infrastructure that provides messaging capabilities" [34]. MOM and libraries or protocols (possibly based on DHT) that provides such services can have very different properties.

When evaluating possible solutions to the problem stated in section 1.2, only open source pub/sub systems with topic-based message filtering semantics and a reputation for high throughput were considered. This chapter will examine some of the previous work done comparing such systems, and motivate why RabbitMQ and Kafka were chosen for the Evaluation.

3.1 RabbitMQ

RabbitMQ is a message broker and AMQP implementation written in Erlang.

RabbitMQ, being an AMQP implementation, is built on the following concepts. *Publishers* are the processes that produce the input to the system in the form of messages. The messages are first received by an *exchange*, which is responsible for routing the message in different ways

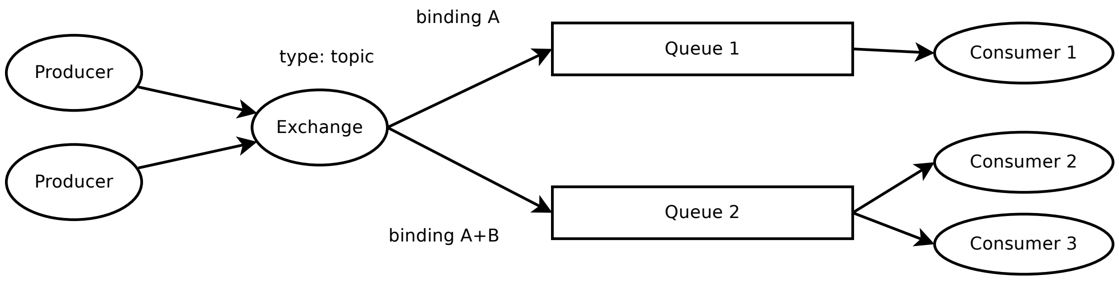


Figure 3.1: RabbitMQ pub/sub system example

depending on which type of exchange it is. One type of exchange is the *topic exchange*, which routes messages based on a message metadata field called *routing key*. Each exchange can be attached to one or more *queues*, that eventually deliver these messages to recipients, or *consumers*, in a First In First Out (FIFO) fashion. One or more consumers can be bound to a queue and receive messages from it, in this case each message will be delivered to only one client. Some queues require clients to *acknowledge* messages that it receives so that in case a consumer crashes the queue can try to deliver the message to a different consumer.

In figure 3.1 the publishers send messages to a topic exchange. Each consumer is bound to a single queue, which is bound to the exchange using binding keys. Each queue can, as previously mentioned, have several bindings to an exchange, matching the consumers' particular interests. In this example, Consumer 1 will receive messages that match the routing key specified in binding *A*, while Client 2 will receive messages of both type *A* and type *B*.

RabbitMQ is very flexible thanks to its core concepts. Messages are sent by *producers* to *exchanges* that route messages to *queues*. The exchanges and queues can be configured to behave in different ways, such as to route messages by their routing key, to broadcast messages to many consumers, or to distribute work to workers.

RabbitMQ has support for clustering. The nodes forming a RabbitMQ are synchronized and all system state except message queues, as well as the data in the system is shared across all nodes. RabbitMQ clusters are guaranteeing full ACID properties, but does not handle network partitions well.[35] Partitions in a RabbitMQ cluster has been proven to potentially cause data loss.[36]

3.2 Kafka

Another message broker is Kafka, although it is not an AMQP implementation. The main point of Kafka is to handle very large amounts of messages in a fast and reliable way. The design of Kafka is heavily inspired by transaction logs.

Kafka was originally developed by LinkedIn and designed for collecting and distributing high volumes of log data. It can be seen as both a log aggregator and a messaging system. Kafka is a topic-based system, where consumers subscribe to topics. The topics are divided into partitions that can be located on different broker nodes in a Kafka cluster. Partitions can be replicated between brokers for fault tolerance, and producers can write to different partitions in parallel. Message ordering is guaranteed, but only for messages within a partition.[37]

Kafka was explicitly designed to support less than a thousand topics[38]. An example of a typical large-scale Kafka deployment is the one used at LinkedIn. In 2012, it was serving 367 topics in total. It is unclear what will happen when Kafka serves a higher number of topics.

A topic in Kafka is split into partitions that correspond to a logical log. Each log consists of a set of segment files that all have approximately the same size. When a message is published to a topic, it is simply appended to the last segment file. To improve performance, changes to the segment are only flushed after a configurable number of messages have been published, or a specific amount of time has passed.[37]

The consensus service Zookeeper[1] is used for decentralized synchronization between brokers in a Kafka cluster. It is used to keep track of brokers and consumers; client subscrip-

[1] `https://zookeeper.apache.org/`

tions, consumer groups and consumption relationships; and for balancing partitions between brokers.[37]

The authors of Kafka conducted an experimental study[37] comparing their system to RabbitMQ version 2.4, and Apache ActiveMQ, which is implements the Java Message Service (JMS) API. The test consisted of publishing 10 million 200 byte messages to a single topic[2] with asynchronous flushing of messages to persistent storage. Kafka supports sending messages in batches, so batch sizes of 1 and 50 messages per batch were used. The messages were then consumed using automatic message acknowledgments, pre-fetching up to 1000 messages at a time.

The results of the above experiment show that, for this one-topic configuration, the producer performance of Kafka was twice that of RabbitMQ when single-message batches were used, and it was eight times faster when a batch size of 50 was used. ActiveMQ's producer performance was orders of magnitude lower. In the consumer test, the Kafka consumer achieved a message consumption rate that was almost four times higher than the RabbitMQ and ActiveMQ consumers.

3.3 RabbitMQ throughput record

An experiment made by Pivotal Software has achieved a throughput of 1 million messages per second using a powerful RabbitMQ cluster, deployed on Google Compute Engine.[39] The nodes used in this experiment had eight virtual CPUs and 30GB of memory each. 32 nodes in total were used to form the RabbitMQ cluster, where 30 nodes were configured to store their state only in RAM, and one node was configured to persist its state to disc. The last node was monitoring the cluster using RabbitMQ's management tool.

[2]Since the authors do not mention the number of topics or partitions used, a single topic configuration was assumed. This seems like a logical choice since it is a throughput-centric experiment and a single topic is the simplest configuration in Kafka as well as RabbitMQ.

Messages were pushed to the cluster using approximately 13000 simultaneous connections. The messages were sent to 186 queues, located at different nodes in the cluster.

The authors of this experiment draw the conclusion that "taking full advantage of a large cluster [...] requires architects to consider how messages are routed within RabbitMQ and the design of the application's so-called message fabric"[39], meaning that this result is largely dependent on the routing scheme. This impressive result shows that when a simple routing scheme is used, RabbitMQ can scale its throughput almost indefinitely. RabbitMQ clusters are, like discussed previously, synchronizing a lot of their state, but a large number of queues can be used since they are not synchronized.

3.4 IoTCloud Message Brokers

The *IoTCloud* platform is a research system that connects smart devices to cloud services for real-time data processing and control. The system uses a layered architecture with a pub/sub layer closest to the devices. The system supports two different message brokers, RabbitMQ and Kafka. As a part of the evaluation of the system, the authors performed benchmarks on both brokers, using pub/sub semantics.[40]

The device endpoints in IoTCloud, called gateways, can either use *shared* channels, where all the devices connected to a gateway are communicating through the same topic; or using *exclusive* channels, where there is one topic per device. This means that the number of topics is either equal to the number of gateways when using shared channels, or to the number of driver instances when using exclusive channels. The team presents three different benchmark results made with RabbitMQ.

They first test the throughput of their server[3], by sending messages to a topic with an increasing message rate in the range $5 - 100$ messages per second. They repeat this for different message sizes, varying from 100 bytes to 500 kB, and measure the latency.

The test showed that for RabbitMQ the latency was stable for messages below 30 ms for all message rates in the interval, for all message sizes up to 300 kB. When 300 kB messages were used, the latency started to increase dramatically for message rates exceeding 50 messages per seconds. The latency also increased for 500 kB messages at 30 messages/s. These numbers indicate that the system had a bandwidth limit at around 120 megabit per second.

The same test was used for Kafka, but with message sizes in the range 100 byte to 70 kB. The latency varied a lot, from around 20 ms to 250 ms. They concluded that "The Kafka latency variation is very high compared to RabbitMQ broker"[40, p. 9] and did not conduct any further benchmarks of Kafka. The reason for the varying latency is not explained.[4]

The scenario described above differs from requirements in section 1.2 in several ways:

- Higher message rates than 100 messages per second are not tested at all
- Throughput is tested mainly by varying message size, instead of message rate
- Only two topics were used in the first test, instead of testing richer filtering semantics using many topics or content based filtering.

[3]The IoTCloud team were using virtual servers with two virtual CPUs, 4GB of memory and 40GB of storage.

[4]Worth noting is that the Kafka configuration item that controls producer batch size is set to 200 messages by default[41], if messages are not flushed explicitly this could account for the latency problems.

4

Design

The design of a messaging system built on top of Riak is described in this chapter. The protocol used by clients connecting to the system will be presented in detail, as well as the overall system architecture and some of the design choices and trade-offs that were made.

4.1 System overview

An overview of the parts of the system can be found in figure 4.1. When a client connects, it is assigned an endpoint to which it will be connected for the remainder of its session. The endpoints are implemented as Erlang nodes that run the logic of the system. Several of these Erlang nodes can be run in parallel on different machines to serve groups of users.

All performance-critical synchronization of these nodes is done through Riak. Other synchronization, such as authentication, will not be discussed further in this thesis, since it can be done using more traditional approaches in a side channel and will not affect the metrics and goals discussed above.

The endpoints are loosely coupled since they use Riak for most of their communication. This serves two purposes. The primary purpose is making the interface between each endpoint and the rest of the system as simple as possible; an endpoint does not have to communicate

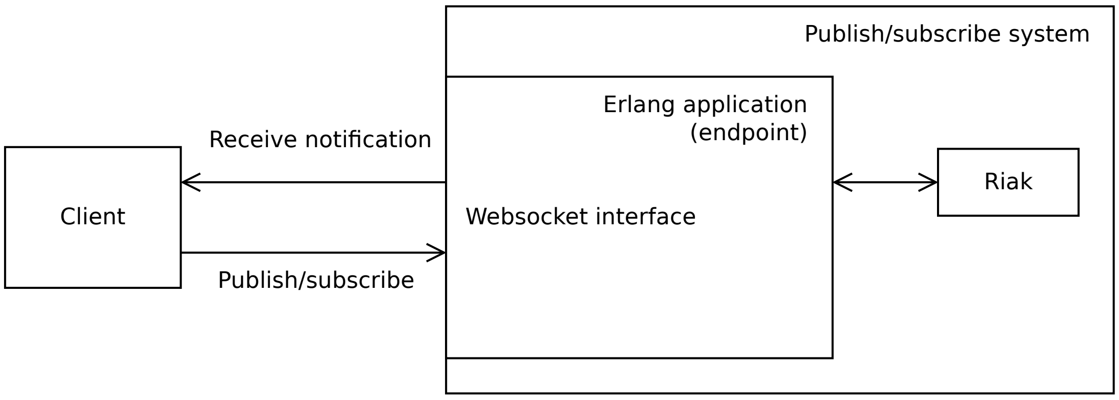


Figure 4.1: An overview of the system's client-server interface

with anything but the Riak cluster, which behaves like a simple key-value store for the most part. Secondly, this design is an instance of a simple layered architecture, see figure 4.2, where each layer only communicates with the layers immediately above and below it. The client communicates only with the endpoint, which communicates with the client and the Riak layer, which only communicates internally and with the endpoint.

The layered architecture has several benefits.[42] Changing the code inside one layer of a layered system does not affect the other layers unless the interface is changed. This means that interfaces should not change, and if they do they should be backwards-compatible. This design also makes components exchangeable. A new layer with the same interfaces as the old one can be implemented. Clients for different platforms could be written, without affecting the endpoint code. It is also theoretically possible to replace Riak with another distributed key-value store, or even another type of data store, in the future.

One of the objectives is to provide high throughput. The layered architecture makes it easier to analyze potential bottlenecks. The maximum throughput for an endpoint, for example, is limited by the throughput of the layer above and beneath it. If only a small number of clients can connect to each endpoint, the throughput will also be lower. Likewise, if more nodes are added to the Riak cluster the potential throughput of the endpoint should also increase.

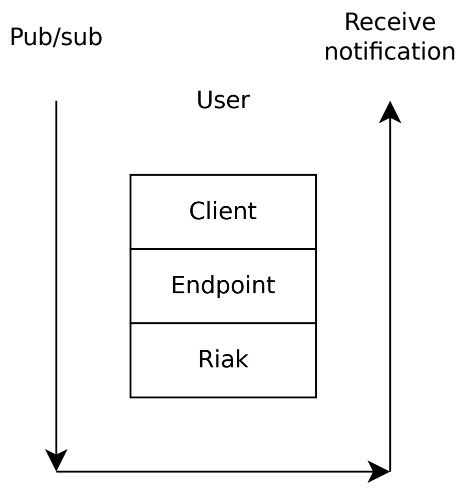


Figure 4.2: The system implements a simple multilayered architecture

4.2 Client interface

From a client perspective, the system supports the following methods:

- Publish: Notify other clients of an observed event.
- Subscribe: Register the client to receive events matching a certain criteria, or from a channel.
- Unsubscribe: Stop receiving events from a topic.
- Receive events: Events that match a client's subscriptions should be sent to the concerned clients without delay.

Client subscriptions are persistent and connected to the client's identifier. After connecting, a client can subscribe or unsubscribe from topics and publish events. Events can also be received at any time, since the client-server connection is full duplex.

The protocol used in this API will be described in the next section.

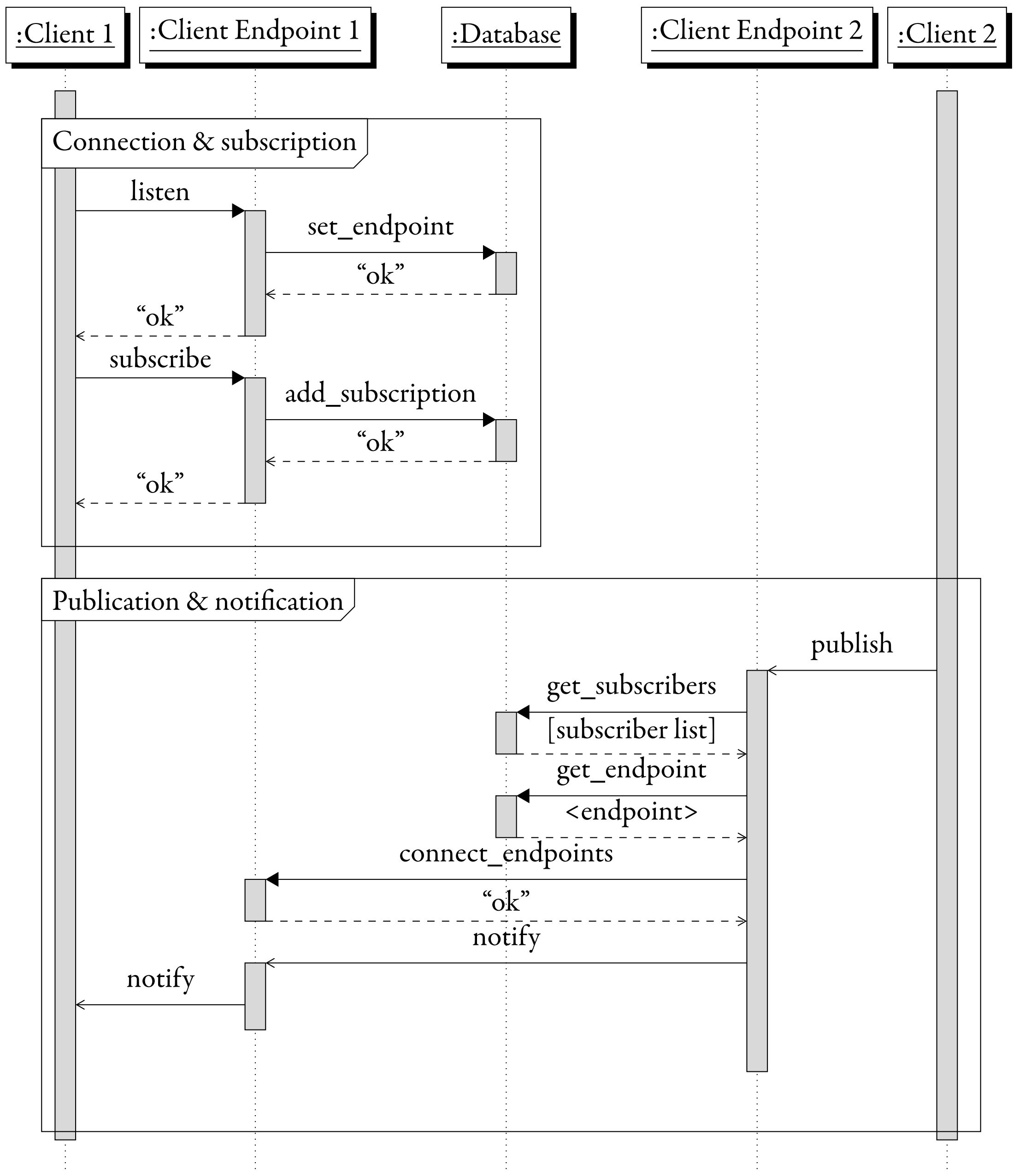


Figure 4.3: Sequence diagram showing the communication between the different entities in the system. Client 1 is the subscriber and Client 2 is the publisher.

4.3 Client-endpoint protocol

A client connects to one of the endpoints of the system. In a production environment the list of endpoints could be synchronized with each client, or they could all be placed behind a load balancer to make all endpoints appear as one. This would allow nodes to be added to the system entirely transparently to the clients.

The endpoint is responsible for maintaining a connection with all the clients attached to it as well as for establishing connections to the other endpoints in the system whenever necessary.

The connection between an endpoint and a client is made over the WebSocket protocol, which allows bidirectional communication using a single TCP connection. Since TCP provides ordering and delivery guarantees, the client-endpoint protocol does so as well. A client connects using a regular HTTP connection, and sends a WebSocket handshake request which allows the endpoint to switch protocols from HTTP to WebSocket. Once the WebSocket connection is established, the client can start sending commands.

The client-endpoint protocol is very simple and based on the exchange of JSON-encoded[1] messages. All commands sent by a client has two mandatory fields: client id and command. The client id is unique and known to each client when they connect, and the command parameter identifies the command type. Specific commands might also include other fields for command-specific data.

```
{
  "command": "listen",
  "client_id": 1
}
```

Code Listing 4.1: Client message: register with the endpoint

[1]JSON: JavaScript Object Notation[43]

A client that wishes to receive broadcast notifications registers with the endpoint using the `listen` command in code listing 4.1. This makes the clients endpoint globally known to other endpoints, which is necessary for routing.

To subscribe to a topic, a client sends the `subscribe` command in code listing 4.2. This command includes a `topic` field, which uniquely identifies the topic that is being subscribed to. The endpoint will create the topic if it does not already exist and add the client to the list of subscribers for the topic.

```
{
  "command": "subscribe"
  "client_id": 1,
  "topic": "Some topic",
}
```

Code Listing 4.2: Client message: subscribe to the topic "Some topic"

The big number of subscriptions supported by the system implies that clients will do many subscriptions. To achieve better performance for clients that subscribe to many topics at once, subscriptions can also be sent in batches. The `subscribe` command has a second form, in which a list of topics is given instead, as seen in code listing 4.3. This dramatically reduces the round-trip overhead cost associated with the first form of the `subscribe` command since only one command is sent per batch, instead of one command per topic.

```
{
  "command": "subscribe"
  "client_id": 1,
  "topic": ["Topic 1", "Topic 2", ...],
}
```

Code Listing 4.3: Client message: subscribe to several topics at once

The publish command takes three extra parameters except `client_id` and `command`: `topic`, `payload` and `timestamp`. The endpoint looks up the subscribers for the topic, then looks up the endpoints for the subscribers and forwards the notification to these endpoints, including the client id of the subscriber. The client's endpoint will forward the notification to the client using a broadcast message similar to the one in code listing 4.7.

The publish command is represented by the "publish" message in the Publication & notification block of figure 4.3.

```
{
  "command": "publish",
  "client_id": 1,
  "topic": "Some topic",
  "payload": "Payload data",
  "timestamp": 1431104020907
}
```

Code Listing 4.4: Client message: publish information to a topic

The endpoint responds to each message sent by a client except for `publish` messages. The response indicates success, like in code listing 4.5 or failure. A success message only has a `result` field, but an error message may contain extra fields to help with debugging.

A client implementation can choose how to handle the different failure modes of each command. A command can be re-issued in case of a missing response, for example. Clients modifying the subscription state on the server will probably find it valuable to receive verification that the issued commands were successful.

```
{
  "result":"success"
}
```

Code Listing 4.5: Endpoint response: the previous command was successful

The `error` field in the example error message in code listing 4.6 contains the error type `bad_state`, which is not very helpful in itself, so the `message` field contain a human-readable error and extra information about the error in the `info` fields. In this case, the error message lets the client know that the required `topic` field was missing from the request.

```
{
  "error": "bad_state",
  "message": "Error",
  "info": "Key 'topic' not specified"
}
```

Code Listing 4.6: Endpoint response: the previous command failed

The `publish` command is the only command that does not expect an acknowledgment response from the server. This trade-off was made because publish and broadcast messages are expected to constitute a majority of all messages handled by the system. Not sending a response to each `publish` command will significantly reduce the number of messages sent by the system. Furthermore, since the time span during which a message is valuable is very short, removing the possibility to re-transmit lost messages does not impede the usefulness of the system since a message that needs to be retransmitted is probably already too old. Re-transmitting on server errors, in this case, could also add stress to an already overloaded system.

The final message type is sent from the endpoint to a client, not as a response to a client message, but to deliver a published message. This is represented by the "notify" message in the Publication & notification block of figure 4.3. The broadcast message has two fields: The

```
{
  "key":"Some topic"
  "broadcast":"Payload data",
}
```

Code Listing 4.7: Endpoint broadcasts notification `key` field is the name of the topic that received the message, and the `broadcast` field contains the payload data, published by another client in the system, possibly attached to a different endpoint.

The WebSocket protocol (section 2.7) does not have a notion of replies, so the application has to keep track of this itself. The client-endpoint protocol described in this section assumes that no message will be sent until a response has been received. A lost message might therefore block the client indefinitely. This can easily be solved by implementing some kind of message id, maybe a sequence number similar to the one used in TCP. This would allow for retransmissions, asynchronous messaging and other features commonly found in transport layer protocols, but this is not yet implemented in the client-endpoint protocol.

4.4 Design summary

In this chapter the design of a messaging system built on top of Riak was outlined.

An interface for connecting clients was specified in the client-endpoint protocol based on WebSockets and JSON. The protocol specifies basic pub/sub actions such as `publish`, `subscribe` and `unsubscribe`.

Clients connecting using the client-endpoint protocol will be assigned one of of the endpoints in a loosely coupled cluster made up by Erlang nodes synchronizing their state using Riak. The goal of this design is to allow scaling the system by adding endpoints to the cluster.

The design makes several trade-offs and optimizations to achieve the goals set forth in section 1.3. The two main design challenges was the requirement to handle a large number of topics, probably more than a million; and to handle an even larger number of subscriptions. One trade-off made in favor of throughput is that the `publish` command is never acknowledged by the endpoint, to save messaging overhead. This could potentially lead to the undetected loss of published messages, but allows a higher throughput. An optimization that was made was to allow multiple topics in a single `subscribe` command – this drastically reduces the number of round-trips when a client needs to subscribe to many topics.

The protocol is not as flexible and robust as general purpose messaging middleware protocols, such as AMQP, since it has a more narrow focus on efficient handling of large number of topics and subscriptions with good throughput and latency. This lack of flexibility and robustness makes it a poor choice for many applications, but a better fit for the case when a large number of clients need to share updates with low latency.

5

Implementation

Chapter 4 described the overall design and architecture of a topic-based pub/sub system. The implementation of that design will be presented in this chapter.

5.1 Ingeborg: Pub/sub built on Riak

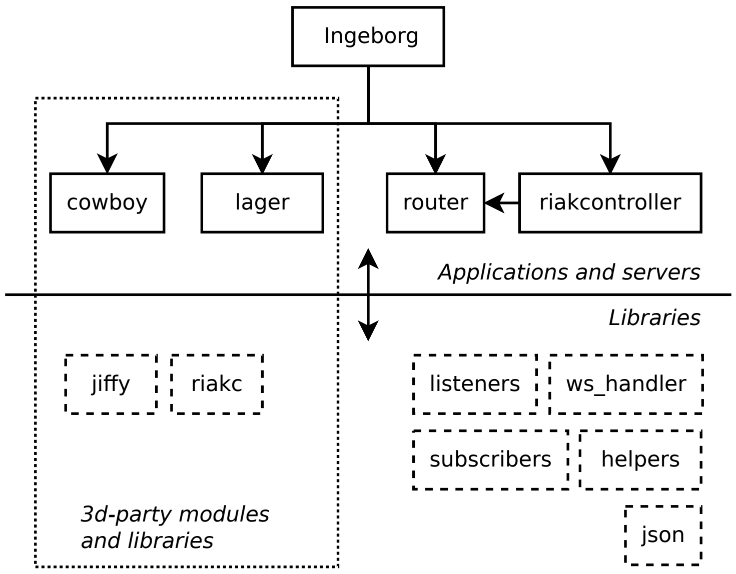


Figure 5.1: An overview of Ingeborg, the pubsub message router built on top of Riak.

The Erlang implementation of the design described in the previous chapter is called Ingeborg. The system consists of several Erlang modules, laid out in figure 5.1.

The third party modules `cowboy`, `lager`, `jiffy` and `riakc` are used. Ingeborg is implemented on top of the Erlang web server framework Cowboy. All logging is handled by the logging framework Lager. JSON-encoding and decoding is handled by Jiffy. These libraries has previously been described in section 2.4.

The `router` and `riakcontroller` modules are implementing most of the core functionality of Ingeborg. The `listeners` and `subscribers` modules are implementing data types for binding clients to endpoints and for handling subscriptions. The `ws_handler` consists of a number of Cowboy callbacks and implements the WebSocket layer of Ingeborg. Finally, the `helpers` and `json` modules contain functions used by many of the other modules.

The purpose of Ingeborg is to allow the client to connect to an endpoint using the WebSocket protocol described above. There can be one or more Ingeborg nodes in a system, they will automatically find each other and synchronize information only when necessary. A client can connect to any endpoint, at which point the `cowboy` server will spawn a new process that sets negotiates a WebSocket connection with the client and then hands over the rest of the session to the `ws_handler` module.

A connected client can start sending commands to the server. To receive messages, the `listen` command needs to be sent. The `ws_handler` module will parse the commands using the `json` helper library, which in turn uses the `jiffy` json parser internally. Once the command has been parsed, a server action is initiated. In the case of the listen command, the `riakcontroller` is asked to update the current endpoint of the calling client to the endpoint that received the command. The logic for how endpoints are stored for each client is controlled by the `listeners` library.

The above is an example of how Riak is used in the application to store and share system state between each Ingeborg node. If a message needs to be routed to a client, by any node, this node can look up which endpoint the client is currently connected to and route the message to it directly.

The `subscribers` library handles lists of subscribers for each topic. The list is stored as the set of the subscribing clients' ids. The set implementation used is a General Balanced Tree[44], which is a balanced binary search tree with automatic re-balancing available in the `gb_set` OTP module. The insertion, deletion and union operators available in the `gb_set` module are efficient even for large sets. A subscription is handled by inserting the client id into the set and an unsubscription by deleting the id. The union operation is used when two conflicting sets are found, which can happen after a network partition since Riak will always allow a write. When Riak discovers a conflict, the application can choose to resolve it, and in this case the union of the conflicting is used to resolve the conflict. In effect, this introduces the possibility that an unsubscription fails, but subscriptions should never fail.

When a client sends a publish command, the endpoint will see if there are any subscribers registered for the key. A new routing process will be spawned for each subscriber to broadcast messages in parallel, processes are cheap in Erlang. The routing process looks up the subscribers endpoint, again using Riak, and forwards the message to it. An endpoint that receives a message routed from another endpoint will look up the clients WebSocket process locally and finally forward the JSON-encoded broadcast message, including the original payload, to the client.

Duplicate detection is not fully implemented, but the idea is to store a unique identifier for the message, maybe $hash(concatenate(key, payload))$, along with a TTL in Riak. When a message is received by an endpoint it will check if the message has been seen recently. If this happens, messages will simply be dropped and the TTL will be increased. If the TTL has expired, the message will be delivered normally. A cache of these recently seen messages and their TTLs could be kept at each endpoint to reduce the number of Riak look-ups.

5.2 Routing algorithm

When messages are broadcasted to a topic the system needs to deliver them to the correct subset of all currently connected clients – the ones that are subscribing to the topic.

An endpoint can receive a broadcasted message in two ways. The first way is from a connected client that broadcasts a message to a topic. The algorithm in code listing 5.1 describes what an endpoint does with each message that it receives. First, all subscribers for the message's topic is fetched. For every client in the list of subscribers, the corresponding endpoint for that client is looked up. If the endpoint is the current endpoint, the message can simply be forwarded directly to the client by looking up which Erlang process that corresponds to the client's connection and forwarding the message to it. If the endpoint is remote, the message is instead forwarded using to the endpoint, together with information about which client should receive it. This is the only instance where two endpoints are communicating directly with each

```
for each $subscriber in lookup_subscriber($message.topic) {
    $endpoint := lookup_endpoint($subscriber)
    if ($endpoint == "localhost") {
        $process := lookup_subscriber_process($subscriber)
        if ($process) {
            broadcast($process, $message)
        } else {
            unregister_endpoint($subscriber)
        }
    } else {
        forward_message($endpoint, $message)
    }
}
```

Code Listing 5.1: Endpoint message routing algorithm other, all of the other lookups above are done using Riak. The intra-endpoint messaging is done using distributed Erlang.

Looking up the subscribers for a topic requires only one look-up, but finding the endpoints for each subscriber requires one look-up each since the subscriber list does not include any endpoint information. The lookup and forward steps are done in parallel using one Erlang process per subscriber.

The second way an endpoint can receive a broadcasted message is from another endpoint. In this case the recipient is always included, so the endpoint looks up the Erlang process corresponding to the connected client and forwards the message to it directly.

When the endpoint queries Riak for the endpoint of a client, the response can be empty. This happens when the client is not connected, so no routing will be attempted. An endpoint might also receive a message for a client that is no longer connected and then the message will simply be dropped and the endpoint information in Riak for the client will be cleared since it is no longer valid.

This simple routing algorithm ensures that messages are delivered to all subscribers for messages broadcasted to any endpoint.

5.3 Riak integration

In chapter 4 and section 5.1 a loosely coupled design was described. In this system, nodes are synchronizing state using Riak and the only direct communication between nodes takes place to forward messages. This section will describe the module `riakcontroller` (see figure 5.1) that handles all communication between an endpoint and Riak.

The `riakcontroller` module that handles communication between Riak and the endpoints exports the following functionality:

- `ping()` – Check whether the connection to Riak is still open.

- `subscribe(client, key)` – Add a `client` to the list of subscribers for a topic identified by `key`.
- `unsubscribe(client, key)` – Remove a `client` from the list of subscribers for a topic identified by `key`.
- `subscribers_lookup(key)` – Fetch the list of subscribers for a topic identified by `key`.
- `listener_register(client, server)` – Update the endpoint (`server`) registered for a `client` and replace any previously assigned endpoint for the `client`.
- `listener_unregister(client)` – Remove any registered endpoint for a `client`.
- `listener_find(client)` – Find the registered endpoint for a `client`, if any.

When an object is read from Riak, sometimes more than one version of the object are returned. This is the result of the consistency model used by Riak. If two Riak nodes are out of synchronization, conflicting updates can be made to the objects stored on the nodes. It is up to the client to resolve these conflicts. This can happen in any update, so the `subscribe`, `unsubscribe`, `listener_register` and `listener_unregister` functions all need to handle conflict resolution.

The `subscribers` module (figure 5.1) handles conflict resolution for the subscribers list. This list is implemented using `gb_set`, an efficient set implementation (see section 2.4). When two different versions of a subscribers list is read for a particular topic, the union of the two sets are used to resolve the conflict. This means that write operations will take precedence over delete operations and a client that subscribes will always succeed, but an unsubscription might be overwritten by an earlier subscription. This resolution model was chosen because it is easy to detect a stale subscription, but hard to detect a missing one. A stale subscription results in broadcasts to a topic that is no longer of interest to the client, and the client can unsubscribe again. A missing subscription, on the other hand, would only be visible if the client regularly verified that it was registered on the expected topics.

The `listeners` module handles conflicts in a different way. The `last_write_wins` Riak setting is used for these values, which means that any existing value will simply be overwritten. In theory, a client could get the wrong endpoint registered, but since these values are updated relatively rarely it should be rare and when it happens, the client can simply reconnect.

6

Evaluation

In this section, the design and results of the benchmarks described in section 6.2 will be presented.

6.1 Test metrics

Three systems were benchmarked. Apache Kafka and RabbitMQ are state of the art message brokers that are known to scale well and handle high loads. The third system is the custom Riak-based system described in chapters 4 and 5. The following sections details which metrics were gathered.

6.1.1 Primary metrics

To test if the goals discussed in section 1.3 are met, load tests with simulated traffic were performed. The users, metadata events and subscriptions were be simulated, but the core parts of the system were tested in a production-like environment.

The key measurements are latency and throughput, as discussed previously. The input variables of the tests are the number of simultaneously connected users and the number of nodes handling these users.

To simplify, each load generator is able to simulate several average users by sending a typical pattern of messages, such as subscriptions and unsubscriptions.

6.1.2 Additional metrics

The following information were also gathered, but in this case they were measured per node:

- CPU load
- Memory usage
- Network utilization
- Disk I/O

6.2 Test strategy

In the previous section, evaluation scenarios were discussed in general. This section will go through how the benchmarks were actually performed.

Since three different systems were tested, the load generator was designed to support three different back-ends. An overview of the test architecture can be seen in figure 6.1. The purpose of the common interface is to make it possible to reuse the same traffic model for the different back-ends without re-implementing common parts such as gathering metrics, service monitoring, logging and rate control.

The benchmark used below is inspired by the jms2009-PS benchmark[45, 46, 47, 48], based on SPECjms2007[49], which is targeted specifically towards testing pub/sub systems. It has over 80 parameters, but the essential ones are:

- Number of topics and queues used

Parameter	**Value**	**Description**
Transactional	False	Is a message sent as a part of a transaction?
Persistent	False	Are messages persisted?
Durable	False	Messages received while a subscriber is offline are stored for later delivery
Queue	False	Whether a queue or topic is used in case of a single consumer
Target destination option	Message type	A single topic per message type is used

Table 6.1: Parameter used in the benchmark, compared to those of jms2009-PS.

- Number of transactional vs. non-transactional messages
- Number of persistent vs. non-persistent messages
- Total traffic per topic and queue
- Complexity of used selectors (filter statements)
- Number of subscribers per topic

The requirements, expressed in terms used in the jms2009-PS benchmark are detailed in table 6.1.

6.2.1 Traffic models

The load generators simulate several clients by sending messages to the system. The messages sent are similar to traffic one could expect to see in a production system.

There are essentially two classes of messaging at the core of the system:

1. Data being published to a topic, or received from a topic
2. Subscriptions and unsubscriptions from topics

The majority of messages are going to be of the first type, so to simplify the benchmarks all subscriptions are being done before the main phase of the benchmark starts and only publish and broadcast messages are exchanged during the test. The subscribe and unsubscribe phase were also recorded separately to get a complete picture of all phases of the test. The main benchmark have three phases:

1. The load generators starts simulating client traffic, by connecting to the system and subscribing to a number of topics.

2. Each load generator publishes messages at increasing rates, until the throughput or latency goals are no longer met. At this point, the load generator should back off by lowering the message rate. This should allow the system to stabilize around the maximum utilization for the current configuration.

3. Finally, the load generators should unsubscribe from all the topics that they are subscribing to. This way, unsubscription messages are also benchmarked, and all features of the system are tested.

The following properties of the benchmark are of extra interest:

- A big number of topics, some very popular and some less popular. There should be a small number of very popular topics, and a large number of topics with only a few subscribers. This should match the use of the system well.

- Messages should be *duplicated*, which in this case means that the message payload is the same for two or more messages from different clients. This is an interesting feature, since the system will need to filter duplicates.

- A large number of simulated clients.

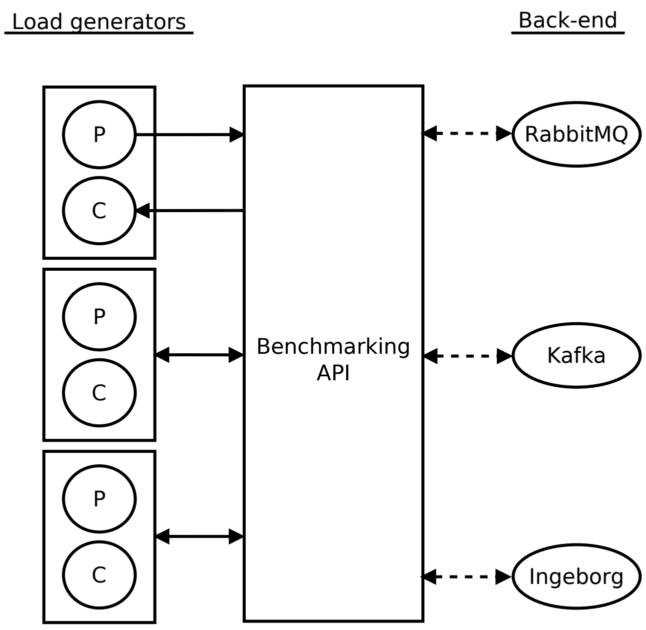


Figure 6.1: Overview of the test architecture, showing consumers and producers communicating with the different systems through a common interface.

6.2.2 Acquiring metrics

This client does not only measure throughput, but it also includes timings of when messages are sent and received. The timing data can be analyzed and the point A to point B latency, as discussed in section 1.3.3, can be calculated on the level of individual messages. The messages are chosen so that a single client will only receive messages that it sent and no time synchronization between clients will be required to calculate the round-trip latency. If messages would have been sent randomly, the system clocks at the load generating nodes would have had to be synchronized with a very high accuracy to compare events logged on different load generators with good precision.

Throughput is calculated from the logs by simply looking at the number of messages that are logged during a certain time.

Other metrics (see section 6.1.2) were also logged. They were logged by a simple script querying the system for the information needed.

To assert that the systems being tested could handle the required loads for a longer period of time, a dynamic load generator was used. The generator starts with a low message rate, and keeps increasing it until the measured latency is above a predefined threshold. At this point, the load is decreased until the latency is below the threshold again. This lets the load generator find a level where the generated load is at a sustainable maximum level. This approach allows one single test run to find the limitations of the system, while showing how the system recuperates after brief periods of high load.

6.2.3 Benchmarking RabbitMQ

Setting up a pub/sub system with RabbitMQ can be done in different ways thanks to the flexibility of AMQP. One possible way is to let producers publish messages to a *topic exchange*. To subscribe, a consumer binds its own queue to the exchange by using a *binding key* which will cause the exchange to route messages to this queue if the *routing key* of the message matches the binding key.

In this case, the routing key can simply be the track. RabbitMQ is supposed to handle large numbers of both queues and bindings well.

6.2.4 Benchmarking Kafka

Kafka is also using topics. A producer publishes messages to a topic and consumers consume them.

Kafka understands both queuing and pub/sub using *consumer groups*. In short, the consumer groups acts like a logical group for one or more consumers. Each consumer that subscribes to a topic in Kafka with a unique consumer group will get a copy of the message.

Kafka stores all published messages whether they are consumed or not, but in this application old messages are useless. Therefore the log retention period should be configured to be as small as possible.

A problem that might occur is that Kafka might use the maximum amount of file handles. This happens with Kafka systems some times when the number of topics are large, since Kafka keeps all log segments open. Ways around this problem might be to add more nodes to the Kafka cluster, or to configure the Kafka nodes to allow a higher number of open files.

6.2.5 Reference system

Many of the web radio streams currently available are broadcasted using a loosely defined protocol that seems to be defined mostly by compatibility with existing clients and servers such as the Shoutcast Server and its open source cousin Icecast. This de facto protocol will simply be called Shoutcast below.

Shoutcast streams are identified by their URL, and a client that wants to consume a stream sends a HTTP request with the specified URL. The server replies with a HTTP-like[1] response followed by some headers that are followed by the response body. The response body can contain either a playlist, a list of other endpoints, or a media stream. In the case of a media stream, the type of the stream is specified in one of the headers and the response body is simply the media stream.

When media files are stored, their metadata is usually stored as a file header (ID3v2[50]) or footer (ID3v1[51]) but since the metadata of a media stream can change after the header has been sent, the metadata needs to be updated some other way. Shoutcast solves this by inserting metadata into the raw media byte-stream at regular intervals.[52] This means that to know

[1]Some implementations will break the HTTP protocol, for example by responding with the status line `ICY 200 OK`, instead of the normal `HTTP 200 OK` response.

which song is currently being broadcasted by such a server, the media stream actually has to be consumed just to receive the interleaved metadata.

The system mentioned in the introduction had a notification service that notified connected clients when a track was played. The server was connected to around 100 selected web radio stations at any time to monitoring their meta data, while the media stream was being discarded[2]. This was becoming costly, since the data traffic for the discarded media streams added up to terabytes every year.

The server, of which the notification service was a part, was a custom server built to be run as single instance. The server could not scale beyond a couple of hundred connected clients at a time, because of the resources needed to monitor the stations mentioned above, and maintaining the state and connections of the connected clients.[3]

6.3 Test platform

The tests were performed using virtual private servers, delivered by DigitalOcean[4]. The node types that were used are described in table 6.2. The *small* node instances were used for network-intensive tasks, and the *medium* were used for services requiring more computing power and memory. Kafka required a lot of memory, so the *large* node type were used for this service.

Kafka version 0.8.1.1 and RabbitMQ version 3.4.4 were used in all the benchmarks below.

[2]The interpretation of Swedish copyright law of the time (2009) was that anyone could make copies of broadcasted music for private use, but not commercially.[53]

[3]There are no exact numbers, since no detailed usage statistics were ever gathered. The numbers presented have been provided by the service's original developers.

[4]`http://www.digitalocean.com/`

[5]666.7 megabits per seconds measured by copying a 1 GiB file with random data between two nodes over the private network interface.

	Small	*Medium*	*Large*
Memory	512 MB	2 GB	4 GB
SSD Disk	20 GB	40 GB	60 GB
Processor Cores	1	2	2
CPU clock rate	2.4 GHz		
Network	Gigabit[5]		
Operating System	Ubuntu 14.04		

Table 6.2: DigitalOcean node specifications. All vCPUs are at 2.4 GHz.

6.4 Traffic benchmark results

In this section, the results of the traffic benchmark described previously are presented. The results for RabbitMQ and Ingeborg are complete, but the benchmark did not yield any useful output in terms of throughput and latency for Kafka.

6.4.1 RabbitMQ

The RabbitMQ benchmark that worked best is shown in figure 6.2. In this scenario, two *small* nodes were configured to generate load against a single-node RabbitMQ system, also running on *small* instances. A *large* Redis node ran alongside the RabbitMQ node, to allow duplicate checking. Traffic was generated using 20 connections, with 1000 topics and 1000 subscriptions per connection.

The reason for running this benchmark with no more than 1000 topics was that running the benchmark with higher number of topics was too unstable and gave unpredictable results in terms of latency spikes, crashes and sudden drops in throughput. To see how the number of topics affected RabbitMQ and the other systems, a simplified benchmark was also implemented, see section 6.5.

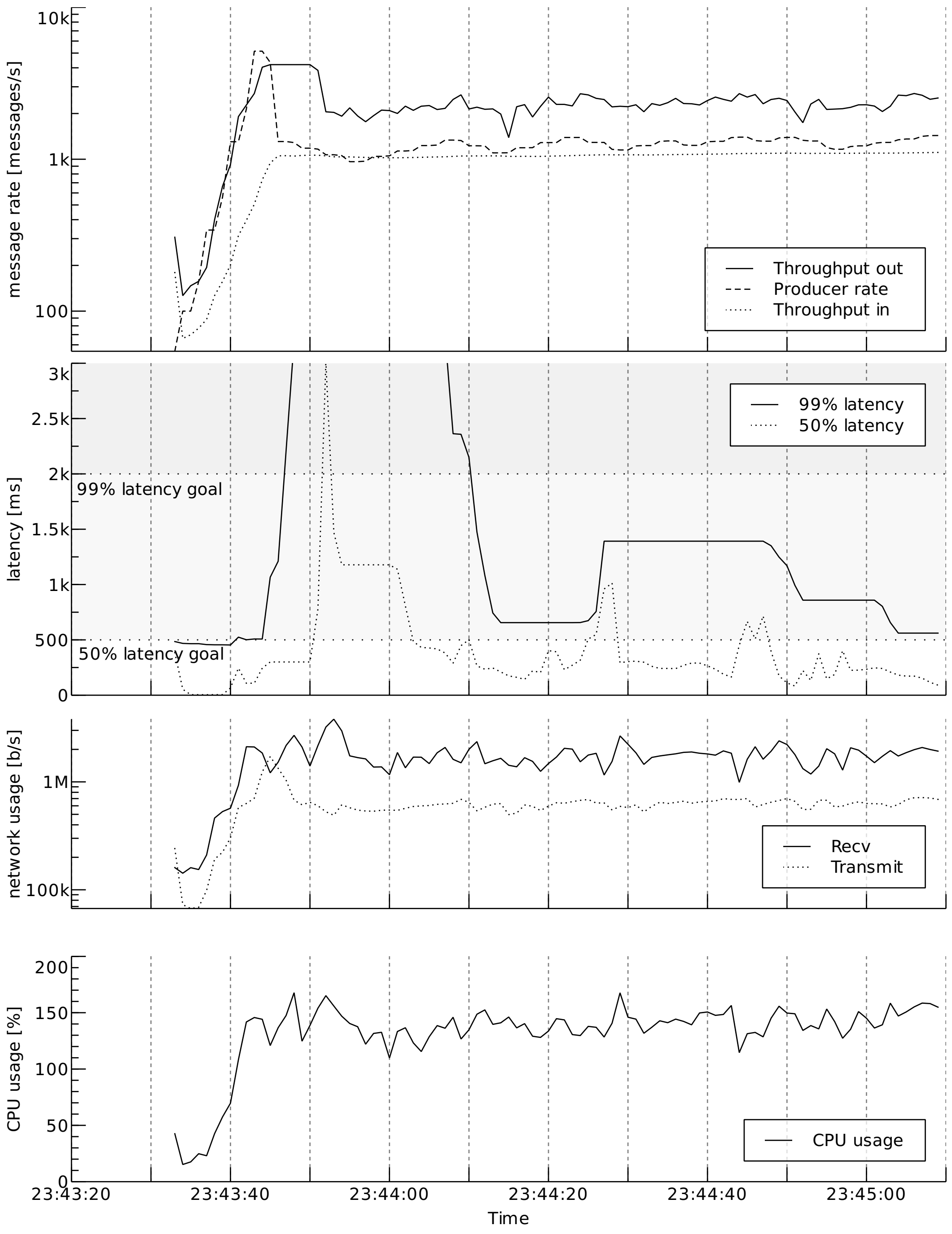


Figure 6.2: Results of RabbitMQ benchmarks. The graphs show message rate, end to end latency, network utilization and CPU usage.

The load generator (see section 6.2.2) stabilizes its produced message rate at around 1300 messages per second. At this point 1000 incoming messages per second are registered by the system, and the system is producing around 30% more messages out, due to broadcasting. This is equivalent to a system handling around 800 simultaneous users, based on the scenario described in section 1.3.2.

The maximum total throughput achieved was 5259 messages per second.

The difference in 300 messages per second is due to duplicate messages being filtered. Another reason for the difference is that if RabbitMQ's message buffer overflows, messages are simply dropped. The rate of dropped messages and duplicates was not measured directly in this test.

6.4.2 Kafka

The test architecture described in section 6.2 did not yield coherent results. The system crashed in the middle of the benchmark with different error messages and exceptions. It seemed like Kafka's performance was degrading with the number of topics. This is consistent with the Kafka designers' stated goal of supporting up to a thousand topics (see section 3.2).

To see how the number of topics affected the performance of Kafka and the other systems, a second benchmark was performed, see section 6.5.

6.4.3 Ingeborg

For evaluating the Ingeborg system, a five node Riak cluster using *small* nodes (see table 6.2) was used to serve as the back-end. Each node in the cluster was also running a local Riak instance. The Riak instances formed the cluster used for synchronizing state between the Ingeborg nodes.

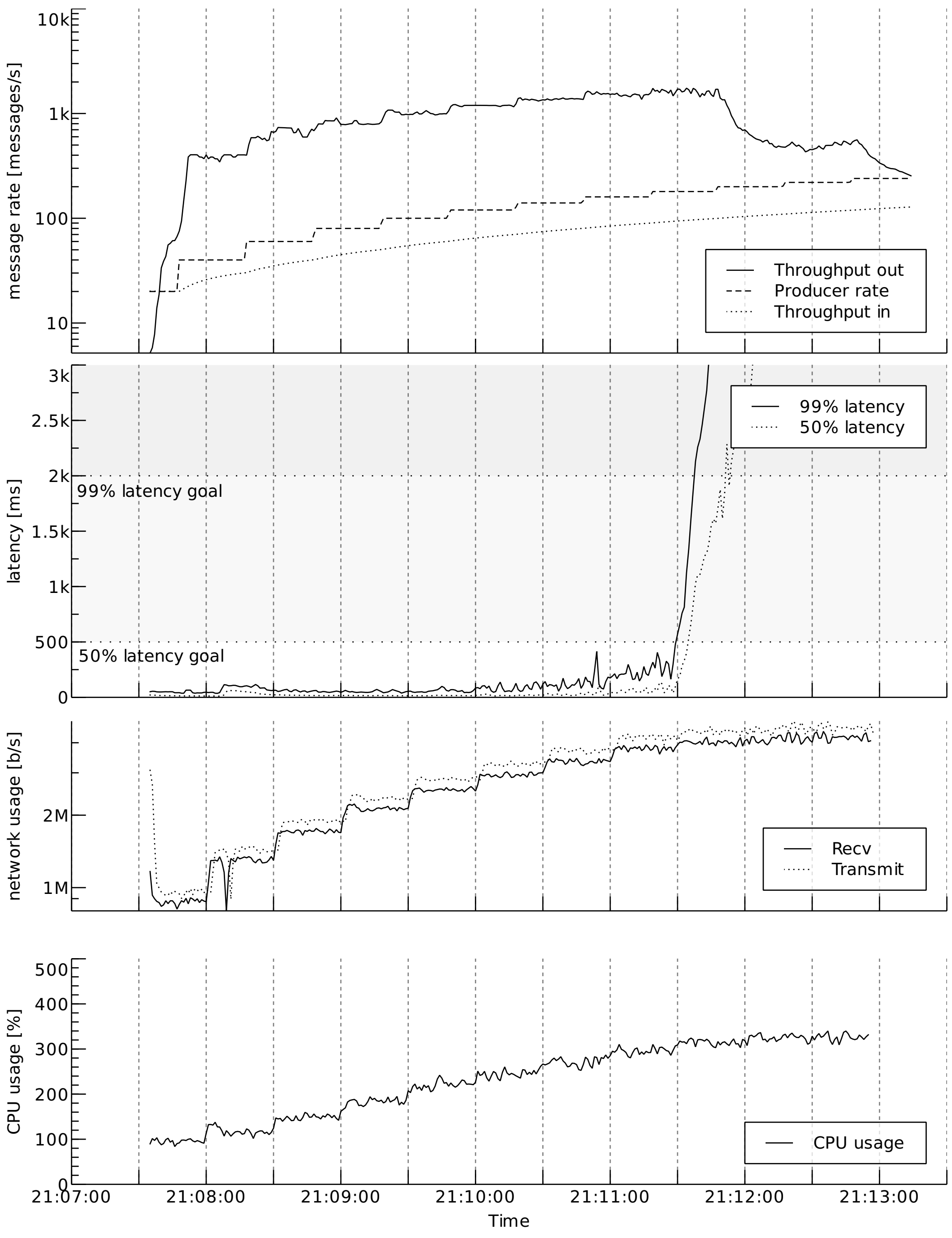


Figure 6.3: Results of Ingeborg benchmarks. The graphs show message rate and end to end latency.

The load generator was running on a *medium* instance. Traffic was generated using 20 connections, with 1000 topics and 1000 subscriptions per connection, the same configuration used for RabbitMQ above. How Ingeborg handles higher number of topics is investigated in section 6.5.

A typical result of this kind of load test is shown shown in figure 6.3. The rate of messages sent to the system is increased gradually until there is a spike in end to end latency. At this point the maximum incoming message rate is about 200 messages per second, and the outgoing message rate is about five times higher. The maximum total throughput achieved, while keeping the latency at an acceptable level, was 1840 messages per second.

The results are fairly stable and shows how Ingeborg reacts when it can no longer handle the amount of incoming messages. When this happens, messages start accumulating in internal buffers, either in Ingeborg or in the Riak back-end. One of the reasons for why the system can no longer keep up is that it spends a high percentage of its time looking for the receiving clients' respective endpoints by repeatedly looking up the client id-endpoint mapping in Riak.

6.5 Topics benchmark

In the traffic benchmarks, described in the previous section 6.4, scaling the number of topics proved to be problematic. The Kafka traffic benchmark (section 6.4.2), only succeeded in showing severe performance issues in this scenario and the RabbitMQ benchmark (section 6.4.1) had to be run with a lower number of topics than initially planned, but showed better results than Ingeborg in terms of throughput even though it was using less hardware. The different amount of hardware in combination with using too few topics makes the result hard to compare.

This prompted a second, less complex benchmark, which addresses a smaller scope than the previous traffic benchmark. Since the large number of topics seemed to be an issue for both Kafka and, to a smaller extent, RabbitMQ – the second benchmark was designed with the intention to show how all three systems handle traffic when an increasing number of topics are used.

The topic benchmark consists of the following steps:

- Reset the system, either by deleting all data or by reinstalling it.
- Synchronously, using a single client, publish a large number of messages to N topics. The number of topics should be large enough to send at least one message to each topic, in a round-robin fashion.
- Connect a single client and subscribe to the same topics that were used for publishing, N in total.
- Consume all the messages, synchronously.

The above steps are performed for different values of N, the number of topics, starting with one and doubling it for each iteration until N reaches 2097152 which should be sufficiently large to stress all three systems.

Three data points are collected for each iteration:

1. The time it takes for a client to connect and subscribe to N topics.
2. Overall throughput while producing.
3. Overall throughput while consuming all the previously produced messages.

A medium node (see table 6.2) was used for this benchmark.

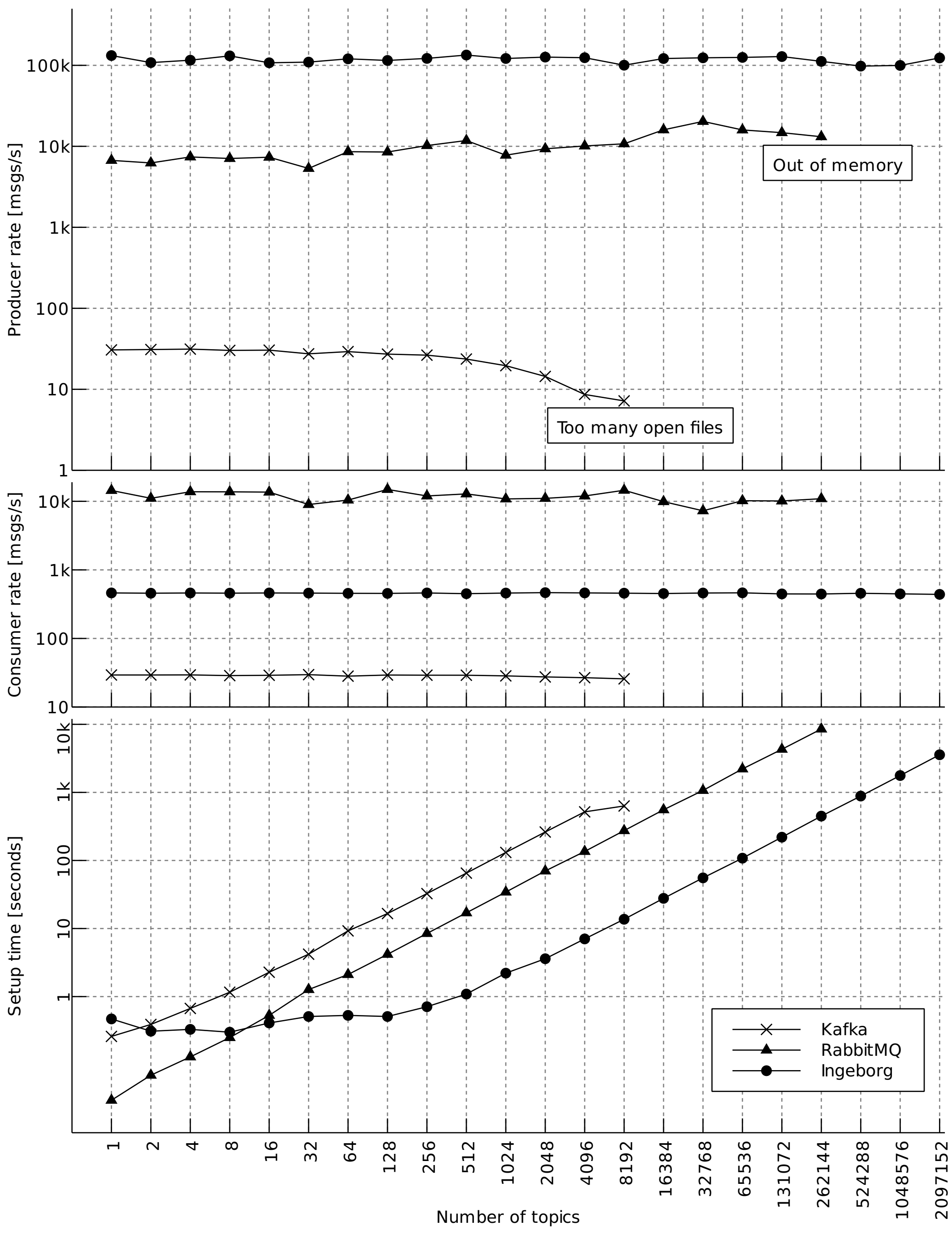


Figure 6.4: Results of the topics benchmark.

6.6 Topics benchmark result

The results of the topics benchmark is presented in figure 6.4. The sub-graphs are showing the producer and consumer throughputs for each iteration, as well as the subscription time.

Ingeborg is able to achieve a very high message rate when producing messages, because no message acknowledgments are used. Reliability is sacrificed, since there is no way to re-transmit a lost message. RabbitMQ is still very fast, even though it is spending some time acknowledging messages. Kafka's poor producer performance have several possible explanations. One is that topics are created on demand in Kafka, so each produced message introduces some overhead when creating a new topic. This overhead might be significant when each message is sent to a new topic. Another factor is that messages are often sent in batches to Kafka, but in this benchmark each message was sent individually.

It must be noted that this is not the intended use of Kafka, as mentioned in section 3.2, the system is designed to handle a limited number of topics, so the results only apply to this particular scenario. A more typical usage pattern would show completely different results. A single Kafka broker is supposed to handle throughputs of hundreds of megabytes per second with thousands of connected clients under normal circumstances.

Each topic in Kafka (see section 3.2) is represented by at least one file in the file system. The consensus service Zookeeper used by Kafka also uses the file system to store information about subscriptions and topics. The system that ran the benchmark allowed one million[6] open file descriptors per process. At 16384 topics, the Kafka system hit this limit and crashed.

RabbitMQ uses files differently, and messages are stored in memory or in persistent storage depending on the configuration. The subscription and topic information uses a significant amount of memory, which became a problem at half a million topics. The RabbitMQ broker used up all available memory at this point, and crashed. Several warnings were raised before

[6]The open file descriptors limit was 1048576. It was set using the command `ulimit -n 1048576`.

the crash and when the configurable memory high water mark was reached no more messages could be published.

For the message rate while consuming, Ingeborg and RabbitMQ have switched positions. Here, RabbitMQ's optimized routing probably works in its favor. Kafka's poor results are probably due to an atypical usage pattern, like before.

Kafka did the worst, again, in terms of the time it took for a client to subscribe to all topics. The on-demand topic creation and Zookeeper synchronization are both probable causes for this. RabbitMQ was three to four times faster than Kafka but 20 times slower than Ingeborg for higher number of topics. RabbitMQ was the fastest for up to eight topics. The ability to subscribe to multiple topics at once was clearly an advantage in this regard for Ingeborg. The Kafka client has a similar option, but it seems like it translates to multiple calls to the broker.

6.7 Summary

Three different pub/sub middleware systems were benchmarked: RabbitMQ, Kafka and Ingeborg. Two separate benchmarks were performed, a more complex traffic benchmark (section 6.4) and a simplified benchmark designed to show how the systems handle a higher number of topics (section 6.5). RabbitMQ achieved the highest throughput for a moderate number of topics, in the traffic benchmark and Ingeborg was the only system that could handle millions of topics without problems.

In the traffic benchmark all systems had their specific problems but RabbitMQ delivered the best results overall, with a maximum total throughput of 5259 messages per second. This is less than three times the throughput of Ingeborg. Kafka could only complete the topics benchmark, because of the excessive overhead when many topics were used. The number of topics used in this benchmark was limited by RabbitMQ's performance, which leads to the topics benchmark.

The topics benchmark showed an advantage of Ingeborg over the other two systems. RabbitMQ's memory consumption and Kafka's many open file descriptors during the topics benchmark confirms that these systems were not designed for this usage pattern.

7

Conclusion

In this section, the findings and results of this thesis will be discussed. The experimental work has shown some of the difficulties in scaling topic-based pub/sub systems. The prototype system's design and implementation gives some ideas for how customized topic-based messaging systems with pub/sub semantics can be implemented.

7.1 Topic-based filtering

Topic-based filtering does not scale without careful design. In the case of RabbitMQ, routing speed decreases with the number of bindings and simply increasing the number of nodes in a RabbitMQ cluster will not automatically make the overall throughput faster. Since a RabbitMQ cluster replicates all data except message queues across its nodes, adding nodes can actually introduce more overhead. Nodes are added to scale the number of queues supported by the cluster, and for reliability.

Getting the complex benchmark described in section 6.2 to work well with different platforms proved difficult. An important part of a benchmark's relevance is that it actually tests a property of the system under test, and not just how well the benchmark itself performs.

The benchmarks performed has shown that both Kafka and RabbitMQ perform very well in a topic-based routing setup when a small number of topics are used but that the scalability is limited when it comes to increasing the number of topics used.

Ingeborg, the system designed and prototyped in this thesis, managed to outperform both Kafka and Ingeborg in this specific scenario for two different metrics. First, by eliminating message acknowledgments it could achieve a high incoming message rate on one hand, sacrificing delivery guarantees on the other hand. In this case this trade-off was acceptable, but in many other applications it is not. Second, the system was designed with a very high number of topics in mind, and kept delivering a stable service long after Kafka and RabbitMQ depleted their resources in different ways.

RabbitMQ has an infinite number of possible configurations. Kafka, on the other hand, is designed to achieve high throughput for many simultaneously connected clients on a moderate number of topics. Therefore, it is important to note that these test results are only valid for setups similar to the one used in this study.

7.2 Optimizing Ingeborg

In the Traffic benchmark results section, it was obvious that RabbitMQ outperformed Ingeborg in a normal traffic scenario by a factor of about 2.85. This should not be surprising, given that RabbitMQ is a mature system that has seen many years in production use across a wide range of environments. How can Ingeborg be improved, then?

One point where Ingeborg could be optimized is in the way message routing is handled. The routing scheme requires each message endpoint to be looked up every time a message is routed. This yields a lot of calls to Riak, so some caching could be done, or the routing mechanism could be redesigned completely. One idea for an improved routing mechanism is to synchronize all broadcasted messages between the Ingeborg nodes and let the clients' own endpoint keep track of the current location of each client locally. This would probably reduce the load on Riak drastically.

The prototyping of this system also highlighted the difficulties in debugging distributed systems. Performance bottlenecks are hard to find unless you have the right data, and this requires monitoring the system state of many machines at once, as well as the program state at each node. Interpreting the monitoring and debugging information can also be difficult due to the sheer volume of information.

7.3 Concluding remarks

There seems to be many dimensions to the word "scalable" in the marketing material and documentation of the available messaging systems. It is not always clear what is meant by the word. RabbitMQ and Kafka both scale well to very high throughputs, for example, but may fail when the system scales along other dimensions, such as the number of simultaneously connected clients; the number of bindings or subscriptions to a topic; the number of queues, exchanges or other system state. There seems to be a need for a common language describing the properties and trade-offs of distributed messaging systems, since the current jargon is often ambiguous or vague.

This thesis set out to show how a pub/sub system could be used to service an application with very high demands not only for throughput and latency, but also with the requirement of a filtering mechanism supporting a large and changing set of subscriptions for each client. In this aspect this thesis has been fruitful, since Ingeborg, the prototype system, could manage at least eight times more topics than the second best system without problems.

The thesis' value is also in showing some of the difficulties and possibilities that exist in topic-based pub/sub systems. It is also clear from the experimental results that a simple system based on RabbitMQ would outperform the reference system (section 6.2.5).

Another positive result is that the prototype system Ingeborg is showing that it is possible to create a new pub/sub system from scratch, on top of a distributed key-value store such as Riak. This allows the designer to consider many trade-offs that are not possible in more generic systems like RabbitMQ, since these systems have to perform under a wide range of possible configurations.

Bibliography

[1] IFPI Digital Music Report 2015: Charting the Path to Sustainable Growth. http://www.ifpi.org/downloads/Digital-Music-Report-2015.pdf, April 2015.

[2] RIAA - Scope of the Problem. http://www.riaa.com/physicalpiracy.php?content_selector=piracy-online-scope-of-the-problem, September 2015.

[3] SFS 1960:729, Lag om upphovsrätt till litterära och konstnärliga verk. Swedish Code of Statuses (Svensk Författningssamling).

[4] Produkter & ersättningsnivåer. http://www.copyswede.se/elektronikbranschen/produkter-och-ersattningsnivaer/, September 2015.

[5] Chilirec (BETA) - Your Free Internet Recorder! https://web.archive.org/web/20071011230336/chilirec.com, October 2007.

[6] Kincaid, J. Dropbox acquires the domain everyone thought it had: Dropbox.com. http://techcrunch.com/2009/10/13/dropbox-acquires-the-domain-everyone-

thought-it-had-dropbox-com/, October
2009.

[7] We've only just begun!
https://news.spotify.com/se/2008/10/07/weve-only-just-begun/, October 7 2008.

[8] Tarkoma, S. *Publish/subscribe systems: design and principles*. John Wiley & Sons, 2012.

[9] Castro, M. et al. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 2002. 20(8):1489–1499.

[10] Rowstron, A. & Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*. Springer, 2001 (pages 329–350).

[11] Van Renesse, R. & Bozdog, A. Willow: DHT, aggregation, and publish/subscribe in one protocol. In *Peer-to-Peer Systems III*, (pages 173–183). Springer, 2005.

[12] Eugster, P.T. et al. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 2003. 35(2):114–131.

[13] Ratnasamy, S. et al. *A scalable content-addressable network*, volume 31. ACM, 2001.

[14] Stoica, I. et al. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 2001. 31(4):149–160.

[15] Zhao, B.Y. et al. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, UC Berkeley, 2001.

[16] Maymounkov, P. & Mazieres, D. Kademlia: A peer-to-peer information system based on the XOR metric. In *Peer-to-Peer Systems*, (pages 53–65). Springer, 2002.

[17] Timpanaro, J.P. et al. BitTorrent's mainline DHT security assessment. In *New Technologies, Mobility and Security (NTMS), 2011 4th IFIP International Conference on*. IEEE, 2011 (pages 1–5).

[18] Aekaterinidis, I. & Triantafillou, P. PastryStrings: A comprehensive content-based publish/subscribe DHT network. In *ICDCS*, volume 6. 2006 (page 23).

[19] Ahulló, J.P., López, P.G. & Gomez Skarmeta, A.F. LightPS: lightweight content-based publish/subscribe for peer-to-peer systems. In *Complex, Intelligent and Software Intensive Systems, 2008. CISIS 2008. International Conference on*. IEEE, 2008 (pages 342–347).

[20] Pujol-Ahullo, J., Garcia-Lopez, P. & Gomez-Skarmeta, A.F. Towards a lightweight content-based publish/subscribe services for peer-to-peer systems. *International Journal of Grid and Utility Computing*, 2009. 1(3):239–251.

[21] DeCandia, G. et al. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, October 2007. 41(6):205–220. ISSN 0163-5980. doi:10.1145/1323293.1294281.

[22] Holcomb, B. NoSQL database in the cloud: Riak on AWS. http://media.amazonwebservices.com/AWS_NoSQL_Riak.pdf, June 2013.

[23] Armstrong, J. The development of Erlang. In *ACM SIGPLAN Notices*, volume 32. ACM, 1997 (pages 196–203).

[24] Armstrong, J. A history of Erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. ACM, 2007 (pages 6–1).

[25] Ekeroth, L. & Hedstrom, P.M. GPRS support nodes. *ERICSSON REV(ENGL ED)*, 2000. 77(3):156–169.

[26] Armstrong, J. Erlang. *Communications of the ACM*, 2010. 53(9):68–75.

[27] Däcker, B. Erlang-a new programming language. *Ericsson Review*, 1993. 70(2):51–57.

[28] Pardo-Castellote, G. OMG data-distribution service: architectural overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*. May 2003 (pages 200–206). doi:10.1109/ICDCSW.2003.1203555.

[29] Extensible and dynamic topic types for DDS. http://www.omg.org/spec/DDS-XTypes/1.1, 2014-11-03. Version 1.1.

[30] Loreto, S. et al. Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP. RFC 6202 (Informational), April 2011.

[31] Fette, I. & Melnikov, A. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[32] Binnig, C. et al. How is the weather tomorrow? towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*. ACM, 2009 (page 9).

[33] Huppler, K. The art of building a good benchmark. In *Performance Evaluation and Benchmarking*, (pages 18–30). Springer, 2009.

[34] Curry, E. Message-oriented middleware. *Middleware for communications*, 2004. (pages 1–28).

[35] RabbitMQ - Clustering Guide. https://www.rabbitmq.com/clustering.html. Retrieved May 25, 2015.

[36] Kingsbury, K. Call me maybe: RabbitMQ. https://aphyr.com/posts/315-call-me-maybe-rabbitmq, June 2014.

[37] Kreps, J., Narkhede, N. & Rao, J. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*. 2011 .

[38] Goodhope, K. et al. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Eng. Bull.*, 2012. 35(2):33–45.

[39] Kuch, J. RabbitMQ Hits One Million Messages Per Second on Google Compute Engine. http://blog.pivotal.io/pivotal/products/rabbitmq-hits-one-million-messages-per-second-on-google-compute-engine, June 2014.

[40] Kamburugamuve, S., Christiansen, L. & Fox, G. A framework for real time processing of sensor data in the cloud. *Journal of Sensors*, 2015. 2015:11. Article ID 468047.

[41] Configuration: Important configuration properties for Kafka broker. http://kafka.apache.org/07/configuration.html. Retrieved May 30, 2015.

[42] Buschmann, F. et al. *Pattern-oriented software architecture: A system of patterns*, volume 1. Wiley, Chichester, 1996. ISBN 9780471958697, 0471958697.

[43] The JSON data interchange format, October 2013. ECMA International, ECMA-404, 1st edition.

[44] Andersson, A. General balanced trees. *Journal of Algorithms*, 1999. 30(1):1–18.

[45] Sachs, K. et al. Benchmarking publish/subscribe-based messaging systems. In *Database Systems for Advanced Applications*. Springer, 2010 (pages 203–214).

[46] Sachs, K. *Performance modeling and benchmarking of event-based systems*. Ph.D. thesis, Technischen Universität Darmstadt, 2010.

[47] Kounev, S. et al. A methodology for performance modeling of distributed event-based systems. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*. IEEE, 2008 (pages 13–22).

[48] Folkerts, E. et al. Benchmarking in the cloud: What it should, can, and cannot be. In *Selected Topics in Performance Evaluation and Benchmarking*, (pages 173–188). Springer, 2013.

[49] Sachs, K. et al. Performance evaluation of message-oriented middleware using the specjms2007 benchmark. *Performance Evaluation*, 2009. 66(8):410–434.

[50] Nilsson, M. ID3 tag version 2.3.0. http://id3.org/d3v2.3.0, Feb 1999. Informal standard.

[51] Id3v1. http://id3.org/ID3v1, October 2012. Retrieved 29 May 2015.

[52] McIntyre, S. SmackFu: Shoutcast Metadata Protocol. http://www.smackfu.com/stuff/programming/shoutcast.html. Retrieved May 29, 2015.

[53] Behdjou, B. Chilirec lanserat i Sverige. *Expressen*, 2009. Jun 10, 2009.

Glossary

AMQP Advanced Message Queuing Protocol vii, ix, 11, 17, 19, 20, 24, 26, 39, 52

BM bit mapping 15

CEP Complex Event Processing 4

Cowboy Erlang web server with websockets support.
Web site: `https://github.com/ninenines/cowboy`
19, 41

DDS Data Distribution Service for Real-Time System 19–21

DHT Distributed Hash Tables viii, 4–6, 12, 14–16, 24

Erlang Programming language designed for distributed systems. Named after Agner Krarup Erlang, a danish mathematician and engineer. vii, viii, 17–19, 24, 30, 38, 41–44

exchange Exchanges are components in the AMQP protocol responsible for receiving messages from publishers, and for forwarding received messages to message queues based on different types of rules. 24, 25, 34, 52, 67

FIFO First In First Out 25

Ingeborg The topic-based publish subscribe messaging system designed, implemented and evaluated in this thesis. Named after Agner Erlang's little sister. 41, 42, 55, 57, 59, 62–64, 66–68

JMS Java Message Service 27

Kademlia Kademlia is a peer-to-peer DHT protocol. Mainline DHT, one of several Kademlia implementations, is used by BitTorrent clients to find peers. 15, 16, 24

Kafka Open source message broker originally developed by LinkedIn.
Web site: `http://kafka.apache.org/`
26–29, 47, 52–55, 57, 59, 60, 62–64, 66, 67

MOM Message Oriented Middleware 24

Open Telecom Platform A collection of libraries, tools and middleware distributed with Erlang. viii, 18

OTP Open Telecom Platform
viii, 18, 42, *Glossary:* Open Telecom Platform

publish-subscribe Messaging pattern where message producers and consumers are loosely coupled. Every sent message in a publish-subscribe system belongs to a message class that consumers can subscribe to. viii, 4

pub/sub publish-subscribe
viii, x, 4, 5, 11–13, 15, 16, 19, 20, 24, 25, 28, 38, 40, 48, 52, 63, 65, 67, 68, *Glossary:* publish-subscribe

QoS Quality of Service 20, 21

queue Message queues are components used by messaging systems for asynchronous communication between processes. Messages sent to a queue can be read by one or more consumers. 5, 11, 25, 28, 48, 49, 52, 65, 67

RabbitMQ Open source message broker implementing AMQP. See section 3.1.
Web site: `http://www.rabbitmq.com`
x, 5, 11, 17, 19, 24–29, 47, 52, 54, 55, 57, 59, 60, 62–68

Riak Distributed key-value data store, inspired by Amazon's Dynamo system
Web site: `http://basho.com/riak/`
6, 16–19, 30, 31, 38, 42–47, 57, 59, 66–68

SEK Swedish krona 2

SOA Service Oriented Architecture 4

topic In publish-subscribe systems, a topic is a named logical channel to which messages are published. xii, 5, 8, 13, 16, 20, 21, 24–29, 32, 35, 36, 38, 40, 42, 44, 48–50, 52, 53, 55, 57, 59, 60, 62–67

USD United States dollar 1

WebSocket WebSocket is a protocol often used for providing bidirectional communication in web browsers. See section 2.7.
xii, 7, 19, 22, 23, 34, 38, 41, 42

List of Figures

List of Tables

List of Code Listings