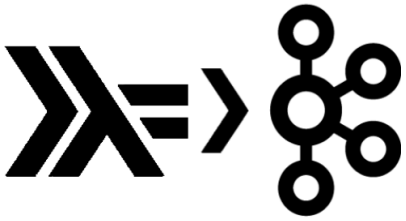


Functional Kafka

Bachelor Thesis
Spring Semester 2015
Computer Science Department
HSR - University of Applied Sciences Rapperswil
http://www.hsr.ch/

Authors: Marc Juchli, Lorenz Wolf
Supervisor: Prof Dr. Josef M. Joller
Expert: Dr. Simon Meier
Scope of work: 12 ECTS (360 hours of work per Student)
Duration: February 16 until June 12, 2015

Preface

Nowadays one can speak of an ongoing revolution in web systems which turns previous demands upside down. Internet companies created a need for activity data in an extremely high volume which has never existed before. And the trend holds on – not only companies with its core business in the cloud also more and more companies with a traditionally fashioned core business are tending to create huge amount of data. This will sooner or later surpass the yet so overwhelming data volume of the internet big five (Apple, Microsoft, Google, Amazon, Facebook). The data those companies generate capture user and server activity. It is at the heart of many internet systems in the domains of advertising, relevance, search, recommendation systems, and security, as well as continuing to fulfil its traditional role in analytics and reporting. Processing of the so called activity data has changed too. The days are gone of nightly batch processing as the only way to produce valuable information – real-time processing dominates the field now, where information is available as soon as data is generated. Being able to serve data to batch- and real-time processing systems requires a central hub that can handle data-load as well as latency. There are many alternative products on the market which are able to provide the necessary capabilities up to a certain point, but among some others (big five) LinkedIn found himself in the unsatisfied position where any existing system was able to process their data load and was scalable at the same time. Thus, they created Kafka, which was soon open sourced under the Apache license, known as Apache Kafka.

Abstract

The aim of this thesis is to provide a summary of the current state of message oriented-middleware and eventually build a message broker in Haskell, adapted from the concepts of Apache Kafka which was originally built at LinkedIn. The implementation shall provide basic functionalities such as producing and consuming messages, with the aim to approximate performance of Apache Kafka in a non-clustered setup. The Apache Kafka Protocol is being used as the underlying wire-protocol and is implemented in a standalone library. On top of the procotol library, a separate client library is provided. Thus, the Haskell Message Broker (HMB) as well as its producer and consumer clients have been successfully proofed as compatible with Apache Kafka.

This thesis first examines the fundamental concepts behind messaging and discloses the needs for message brokers. In a second stage of this technology research, the purpose of event-streaming is described, containing a comparison of batch and stream processing by explaining the differences in their nature. Finally the concept and features of Apache Kafka is presented. Insights into the HMB implementation is provided in the technical report and is split into two stages. At first, the protocol and client library is introduced. Subsequently the broker implementation is explained including its capabilities as well as the provided set of features. After all, HMB is applied to a benchmark against Apache Kafka.

The results of this proof of concept show that Haskell is well suited to build messaging applications as well as implementing protocols based on context free grammars. The from HMB provided performance hit the one of Apache Kafka for transmission of larger message sizes during the benchmark. For the most tested scenarios the performance suffers as HMB is not sufficiently optimized yet. However, the Haskell Message Broker is a well established basis of a state-of-the-art message broker implementation. The authors recommend to apply further optimization techniques as well as extending the feature-set before any other use.

Acknowledgements

Many thanks to our supervisor Prof Dr. Josef M. Joller, who provided us the opportunity to perform a thesis which is about programming a real world application in Haskell. It was our desire to invest time in learning this functional programming language as it has never been an essential part of our studies. By including messaging and Apache Kafka as basic conditions, he defined a very interesting and challenging intent for our thesis. He motivated us to work hard on the given goals.

Many thanks to our expert Dr. Simon Meier, who also was our contact for Haskell specific questions. We are thankful for his inputs related to our code especially the detailed code review he performed. We were very pleased that he gave us the opportunity to participate at the ZuriHac and thus to get in touch with other Haskell experts.

Table of Contents

Part I.

Introduction

1. Management Summary

Introduction

This thesis aims to adapt the concept of Apache Kafka and build a messaging system, namely Haskell Message Broker (HMB), using the functional programming language Haskell. The focus, hereby, lies on the implementation of a stable and scalable server application system as well as on building a log subsystem, which in Apache Kafka is considered to be the most important feature. As Kafka comes with its own wire-protocol, a part of this thesis will focus on a fully compatible implementation of the Apache Kafka Protocol in Haskell, including a client library allowing Haskell applications to take use of the protocol implementation and communicate with Apache Kafka or HMB.

Approach

Becoming familiar with the state of the art in messaging, especially event streaming, was inevitable for us to be able to create our own messaging system. As an essential part of this prestudy we analysed the approach and functionality of Apache Kafka. During the first third of our work, we became familiar with the functional programming paradigm and the programming language Haskell, which we studied intensively. After the prestudy phase, we developed an architectural prototype to demonstrate some very basic functionality of a message broker. With this in our hands, we then continued working out the details for the protocol implementation and the server application. A code review by expert Simon Meier helped us to tweak our code and improve its efficiency. Finally, we tested our system under heavy load in order to optimize the performance of our application.

Results

The first result of this thesis is the prestudy documentation. It provides an insight into messaging fundamentals and takes a closer look at Apache Kafka and related topics. It could potentially be used as an academic amendment for existing lectures. Another result is the implementation of the Kafka protocol in Haskell. The design decision of separating protocol related code from the broker implementation led to a stand-alone product, which can be used as a library for different projects. The open sourced code has already been highly praised by the Haskell community and found a handful of contributors who helped uncovering minor issues. The resulting broker application provides a server with basic networking functionality and persisting messages. It adapts some features from Apache Kafka and provides the ability to produce and consume data. It supports Kafka clients as it is based on the protocol implementation mentioned above. Simple console clients are provided to demonstrate functionality.

2. Task Description

Supervisor

- Prof. Dr. Josef M. Joller, Professor of Computer Science

Problem Description

The functional programming paradigm becomes more and more integrated within classical programming languages: Lambdas, streaming API just to name a few. Languages such as Scala with its rich type system are only understandable if the developer brings extensive knowledge of a strict functional language with. The past shows that only highly experienced developer are capable of producing usable and maintainable Scala code in industry projects.

Due to the supervision, the MAS course as well as the Multi-Paradigm partial module in the compiler course are based on Haskell.

Assignment of Tasks

One goal of this thesis is to work out an overview of the current state of message-oriented-middleware (MOM), especially approaches of event-streaming.

> Essential outcome of this part is the specification of the in Scala written message broker "Kafka" which was developed at LinkedIn and stands now under the Apache license. Kafka claims to be *fast*, *reliable*, *scalable*, *durable* and *distributed by design* (documentation of Apache Kafka: http://kafka.apache.org/).

A further goal is the definition and implementation of the communication protocol ("wire-protocol") of Kafka using Haskell.

> This part of the thesis is essential since differently (programming language, operating system) developed clients will communication with Kafka.

The third goal will be the implementation of the basic functionalities of Kafka in Haskell.

> Since Kafka is strongly decentralized by design, it uses "ZooKeeper" (Apache) on the underlying layers. This functionality (multi cluster) is beyond the scope of this bachelors thesis. This thesis will be reduced to only one cluster.

An important goal is performance comparison of the implementation in Haskell against the one in Scala.

Additionally Kafka has to be installed as well as a benchmark has to be defined.

Realization

In order to work on this assignment, familiarization in the technical basics of message brokers as well as the programming language Haskell is required.

Solid skills in programming and the will to work within new domains of knowledge is required as well.

Because of the variable spectrum of this assignment the topic is perfectly suitable to be handled by two students.

Usually the meetings with the supervisor will be take place on a weekly basis. Additional meetings will be requested by the students on their needs. Every meeting has to be planed using an agenda as well as documented using a protocol, which has to be sent to the supervisor.

To proceed this assignment a project schedule has to be created. Thereby the focus lies on continuous and transparent work steps. Corresponding the milestones, early versions of this thesis has to be provided. The students will then receive feedback. The final grade will be provided based on the version of the implementation as well as documentation which were delivered at due date.

Documentation and Delivery

Since the results are intended to be used in further studies, the priority is set to the completeness as well as the quality (grammar and graphics) of the documentation.

The documentation regarding the project planning and tracking has to be proceeded by the guideline of the computer science department. The detail requirements regarding the documentation of research as well as the results of the implementation will be defined according to the concrete task schedule.

The complete documentation has to be handed as three issues of a CD. Additionally to the documentation the following has to be provided:

- a poster to demonstrate the work
- all data which is required to understand the results and files (source code, build scripts, test code, test data etc.)

Schedule

Spring Semester 2015	Beginning of the thesis. Assignment description is provided by the supervisor.
1. Week	Kick-off Meeting.
2. Week	Delivery of a project plan (draft), including a potential list of hardware to be provided.
4. Week	Delivery of early technology research report as well as a detailed proposal of a working plan. Fixing of the project plan together with the supervisor.
6. Week	Delivery of an implementation proposal regarding the experimental environment. Scope of functionality and time estimate coordination and definition together with the supervisor.
7. Week	Delivery of requirement and domain analysis for the experimental environment.
10. Week	Review meeting regarding software design of experimental environment.
13. Week	Presentation of the current implementation state.
Early in June 2015	Delivery of an abstract for the thesis booklet as well as the A0-poster to be reviewed by the supervisor.
Mid of June 2015	Delivery of the report as well as the final poster to the supervisor.

Table 2.1.: Schedule

The detailed dates will be provided by the computer science department.

References

1 Apache Kafka: http://kafka.apache.org
2 Apache ZooKeeper: http://zookeeper.apache.org
3 Haskell:
Hutton Programming Haskell http://www.cs.nott.ac.uk/~gmh/book.html
Miran Lipovača; Learn you a Haskell http://leanyouahaskell.com
Bryan O'Sullivan, Don Stewart, and John Goerzen Real World Haskell http://book.realworldhaskell.org

Part II.

Technology Research

3. Motivation

In distributed systems and the involved technologies, a significant amount of terms and definitions have been given in the literature. The first part of this technology research aims to achieve more clarity in this confusing and oftentimes ambiguous *jungle of terminology*, where the focus lies on topics that are related to Apache Kafka. After defining basic concepts for referencing during this thesis, we go deeper into the technology and components of Apache Kafka itself.

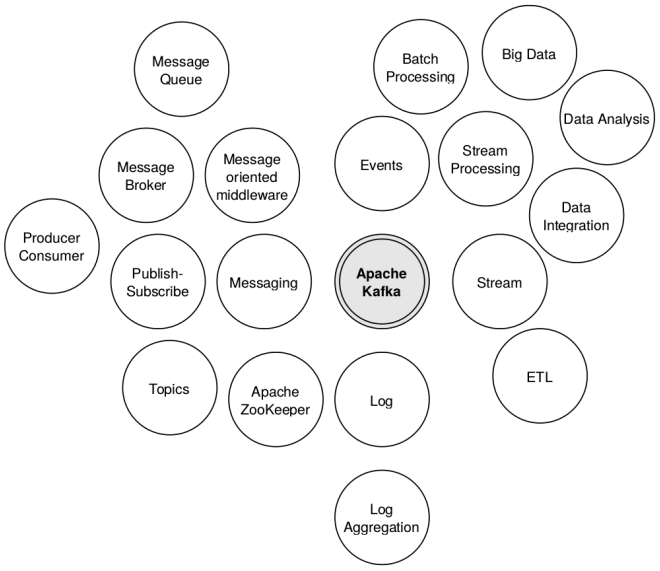


Figure 3.1.: *Jungle of Terminology* related to Apache Kafka

4. Introduction to Messaging

This chapter will at first provide a basic introduction to messaging and will uncover the fallacies of distributed systems, which will lead to the need for a Message-Oriented-Middleware. The primary focus is set on a detailed analysis of a message broker system, which is related to the broker pattern. During this chapter, other terms will be declared which will play an important role to understand the capabilities of Apache Kafka.

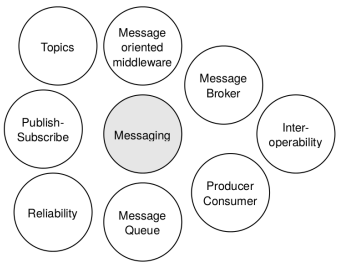


Figure 4.1.: Messaging terms

4.1. What is Messaging?

A distributed system consists of a collection of multiple autonomous components, connected through a network, which enables to coordinate their activities and to share resources. To a user, these systems appear as a single, integrated computing facility. Unlike centralised systems, distributed systems have their advantages in economics, performance and scalability and reliability. In fact, this approach allows to build reliable applications, scaled using low cost computers, and thus can serve massive performance needs whether on demand or continuously. Although distributed systems have many advantages, compared to local applications, they also bring many challenges, as explained in detail later on, which have to be considered by developers.[20][31]

Messaging integrates distributed applications to an overall system by providing message queues for communication between them (see figure 4.2). One application can publish its messages to the queue and another application can asynchronously read from it at any time. A message is just a simple and independent unit with a data structure which contains a meta header and the transmitted information. The messaging technology enables high-speed, asynchronous, program-to-program communication with reliable delivery.

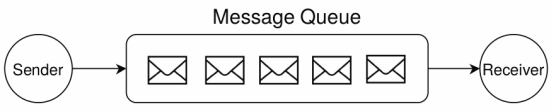


Figure 4.2.: Basic principle of a message queue

Peter Deutsch originally defined fundamental fallacies of distributed computing which still are relevant today.[12] Messaging simplifies the development of distributed systems by mainly facing the following:

1. **Fallacy: The network is reliable**

   There are always plenty of things that can go wrong in a network, like power failures, bad connectivity or natural disasters. Therefore a network must never be seen as fail-safe. One always has to assume that sent data is possibly getting lost on the wire. A messaging system runs with an underlying middleware (4.2) which deals with failures on each side of the communication. This allows to get knowledge about which messages has been successfully delivered and which need to be redelivered. Thus, the original sender never has the guarantee that the sent message has arrived at the desired destination, but it always has the confirmation of the middleware that the data has been put in a message queue. In case of failure the messaging system can initiate appropriate actions to delegate a failed transmission.

2. **Fallacy: Bandwidth is infinite and the Latency is zero**

   Since communicating over network is significantly slower than local operations, a distributed system has to deal with delays and interruptions of any length. Messaging supports a storage capacity which leads to a persistent way of communication where it is not required that the collaborating endpoints are active during the entire transmission of a message. This loose coupling in time is achieved by working with queues as data structure (first-in-first-out).

3. **Fallacy: Topology doesn't change**

   Applications, systems and topologies change over time, and in most scenarios this will often affect the collaborating components in the distributed network too. Therefore distributed nodes should be decoupled as far as possible. Messaging tries to achieved this with the underlying middleware (4.2). The sender and receiver do not need to know whether or not to communicate, nor do they need to know the other side directly. They always communicate via the middleware software. In case of significant changes, only the interface to the messaging system needs to be updated.

4. **Fallacy: The network is homogeneous**

   In the real world, applications are always different. In distributed systems data needs to be transmitted between nodes which use different programming languages, operating platforms, and data formats. This is called interoperability which actually means that systems based on different technologies can collaborate together by using the same standards or protocols. Messaging provides either a common interface or a protocol on the wire which can be implemented by various applications. When using a messaging system in an enterprise environment, it is much more efficient to integrate a new application to the existing data flow instead of implementing a new interface for another technology. The newly introduced component can be linked to the existing platform by just using the underlying messaging protocol.

   In the past, several companies like IBM or Microsoft developed their own proprietary standards and protocols for asynchronous messaging systems – probably to keep it locked in their customer base. In June 2001 the Java Message Service API (JMS) was released as the best-known standard for messaging systems. However, it is only an interface without a specific protocol, JMS implementations need to define their

own. Fortunately in June 2006 a pool of multiple companies for network technologies defined the Advanced Message Queueing Protocol (AMQP) as an open standard for an interoperable messaging protocol. [33]

[22][13]

4.2. Message Oriented Middleware

Because networks are not inherently reliable, messaging would never be efficient without a systems that manages the capabilities of messaging by acting as mediator. To introduce an intermediate component between distributed applications may be a very good choice as it would reduce the close coupling of communication styles such as sockets-based or RMI/RPC. This additional component is known as the messaging system or message-oriented middleware (MOM) and is all about passing messages in any format from one application to another where the parties do not need to knows each other directly. [31]

The MOM provides a specific API for both the sender and the receiver:

Put	Append a message to a specific queue
Get	Block until the specific queue is nonempty, and remove the first message
Poll	Check a specific queue for messages, and remove the first
Notify	Install a handler to be called when a message is put into a specified queue

Table 4.1.: Basic API for message queue implementation [31]

In a basic scenario including an application A being the sender of a message and an application B being the receiver, the MOM – running either on the node itself or dedicated in the local network – holds a local queue to store incoming messages and manages an address-lookup database for mapping destination names to network locations. Overall, the flow of a message sent from A to B will be as follows: first, the sender packs its data into the appropriate message format (1) and puts the message in a local queue (2). The MOM then delivers the message over the network to the local queue of the receiver (3). Finally, the receiver system fetches the message and stores it in its own local queue (4) to further unpack and finally get the transmitted data (5).

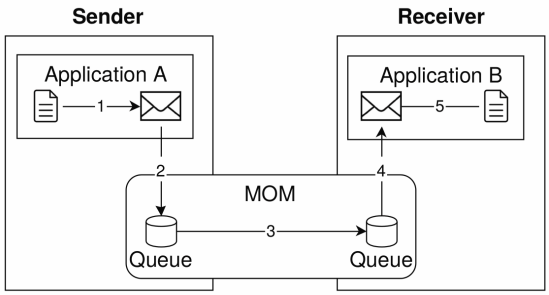


Figure 4.3.: Schematic flow of communication in a basic messaging system

4.3. Topologies

4.3.1. Point-To-Point Connection

A point-to-point connection is the simplest architecture for communicating in a messaging system. It ensures that only one node receives a specific message by setting up particular connections between the nodes. This kind of communication works well in an environment with only a few systems which want to collaborate and use the advantages of messaging. But if there are more nodes, the local mapping of a queue name to a foreign network location implies that each system needs to hold the addresses of all the other nodes that it wants to communicate with. When a target address changes, all the applications that communicate with the changed target must be updated.

In most scenarios there is also a need for transforming messages to a different format where these implementations have to be made on every node, which leads to duplicating code. In an architecture with many collaborating applications, point-to-point messaging can end up in a mesh of connections which is hard to maintain. For more complex integration solutions, a centralised solution whereas the MOM runs on a location-independent platform is required. This is where the central hub (broker) comes into game. [29]

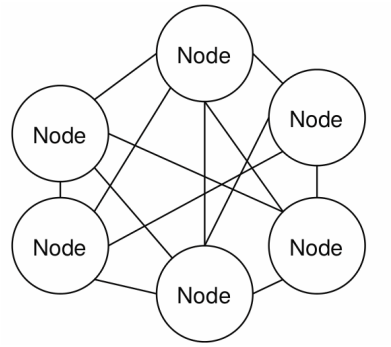


Figure 4.4.: Mesh of multiple point-to-point connections

4.3.2. Central Hub (Message Broker)

A messaging system with point-to-point connections can decouple two systems by using a managed queue on each side of a collaboration. But there is still the need of configuring the endpoints on each side. In general, a message broker is a dedicated component which decouples source and target systems by assuming full responsibility for coordinating communication between all connected nodes. The main tasks of a message broker are the dynamic registration of endpoints, determining the location of a target system (a.k.a. routing) and performing the communication as well as the transformation of a message from one format to another. Furthermore the broker can provide temporal persistence of messages. [29]

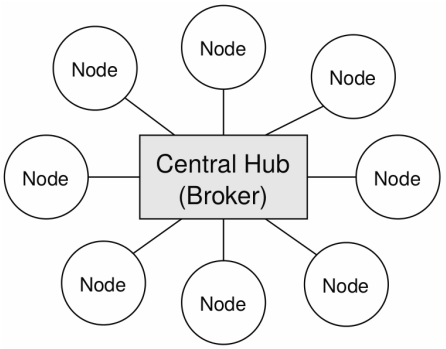


Figure 4.5.: Mesh of multiple point-to-point connections

4.4. Message Broker in Detail

4.4.1. The Broker Pattern

As the initial point for the detailed definition of a message broker we take the *Broker Pattern* of *Patterns of Software Architecture Volume 1* which can be used to structure distributed systems with decoupled components. Basically it distinguishes between two types of brokers, where the *Direct Broker* performs a initialization of a connection but the following communication is only between the two endpoints. For messaging, the second variant, *Indirect Broker*, is a lot more interesting because it maintains all of the communication between the nodes at every time. So let us define the message broker as a refinement of the Indirect Broker Pattern for message-based communication.[20]

The *Broker Pattern* consists of five types of participating components, which we can adapt for a message based environment by using the common terms.

Producer (adapted from Client component)
: Applications that either generate data for later processing or access a service from another client in the messaging system. In both cases the producer pushes its messages to the broker and in general does not care about it anymore.

Producer API (adapted from Client-Side-Proxy)
: Structural component which encapsulates messaging-specific functionality from the rest of the client code. Thereby, a call from the producer can be transformed to a message in the right format. The producer API directly communicates with the broker API.

Broker
: Central component that acts as a mediator and is responsible for the transmission of messages from multiple producers to multiple consumers. A broker offers an API to the messaging clients, both for consumer and producer. It also must have kind of a directory for locating registered consumers.

Consumer (adapted from Server component)
: Basically the consumer is an endpoint which receives and processes a message for further activities. An *Event Driven Consumer* handles incoming messages automatically as they are pushed from the broker. A *Polling Consumer* explicitly gets messages when it wants to receive them from the broker. If a consumer provides a service to other clients, it also can act as a producer for sending back messages as response.

Consumer API (adapted from Server Side Proxy)

Structural component which encapsulates messaging-specific functionality from the rest of the client code. Based on incoming messages it calls services within the consumer component.

4.4.2. Consumption Model

A fundamental design decision of a message broker is whether consumers should pull data from brokers or brokers should push data to consumers. This has a direct impact on the performance of a broker system. In a push-based environment the consumers can be the bottleneck when its consumption rate falls below the rate of production. In a pull-based system, consumers which fall back can catch up at any time, the risk of bottleneck lies at the broker, thus a message broker should scale well.

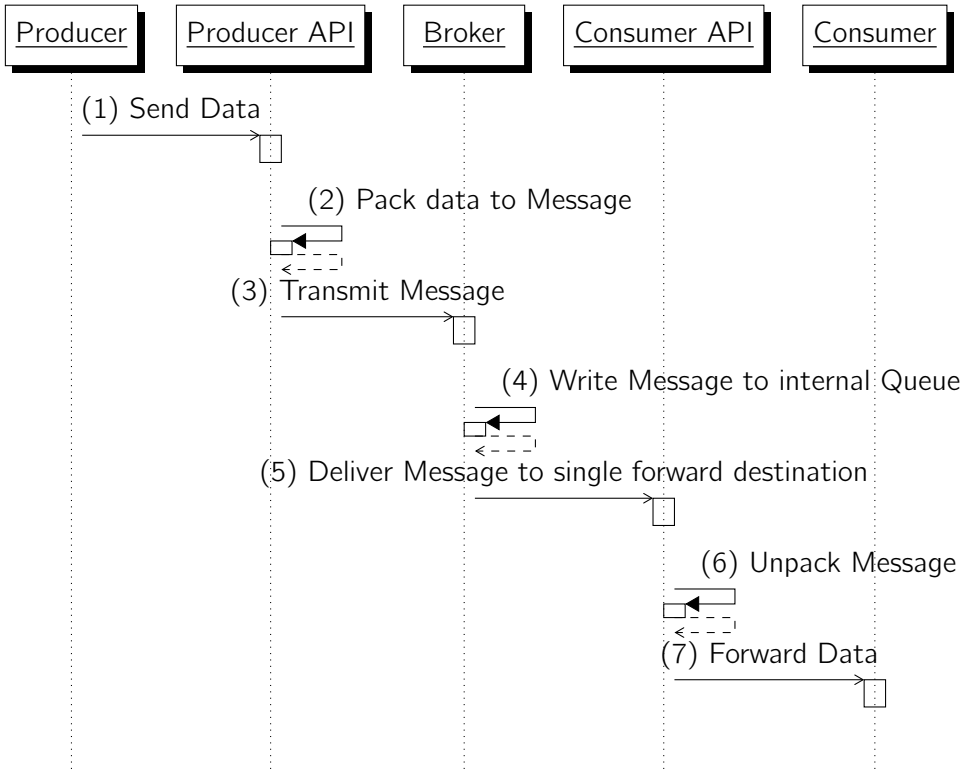


Figure 4.6.: Basic flow of a Broker with event based consumption

On the other hand, in a pull-based environment a consumer never knows whether a message is ready for consumption or not. In the worst case this ends up with a read loop where the consumer constantly needs to check the queue.

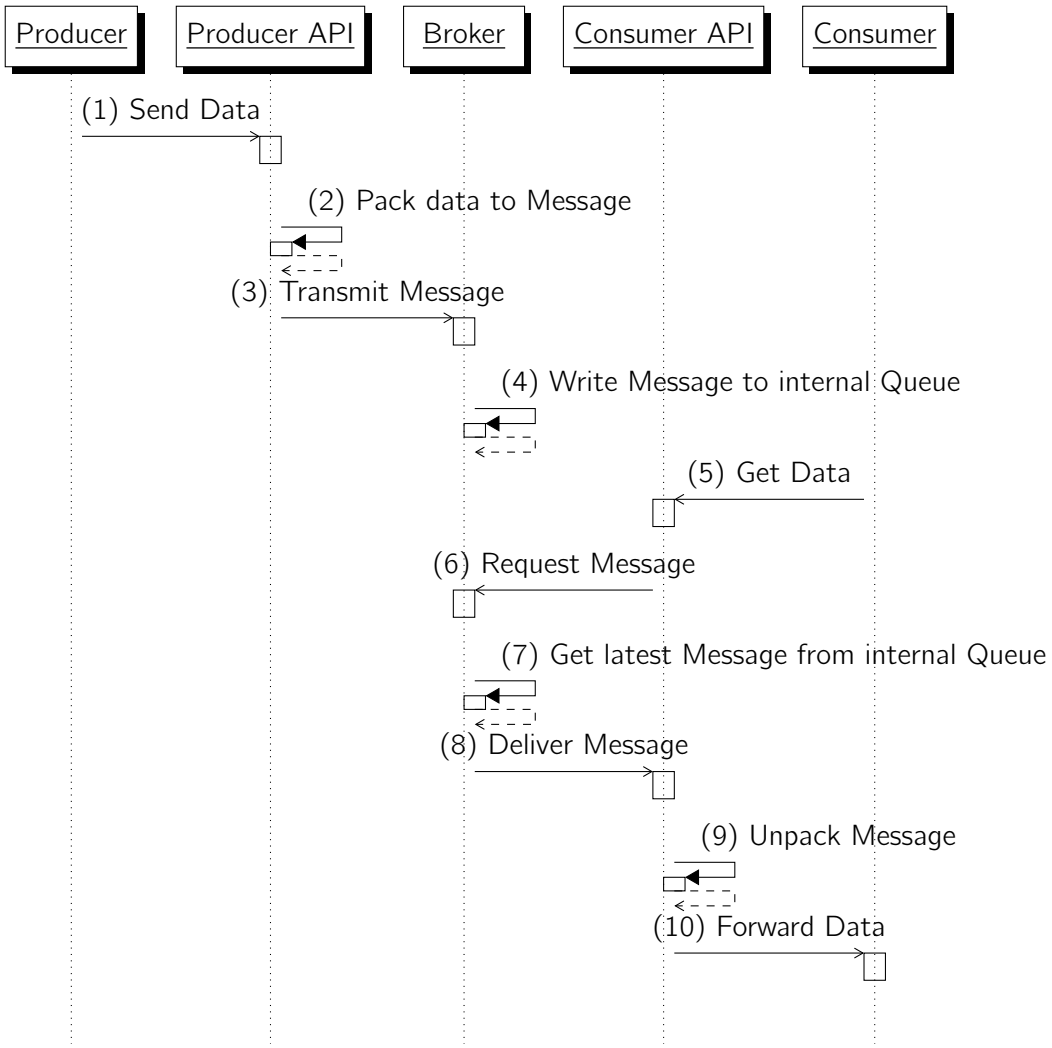


Figure 4.7.: Basic flow of a Broker with polling-based consumption

4.4.3. Topic and Subscription Model

Message Broker systems are persisting incoming messages in an enhanced type of a message queue, named topic. A topic allows the broker to deliver a message to multiple independent consumers. Sending messages to the broker in the form of publishing to a specific topic and on the other hand receiving messages only for the specified topic, is called publish/subscribe. In contrast to a one-to-one queue, the message broker requires the abilities to match messages on an application-based level to act as a gateway for topic-related messages. Based on the informations provided within the messages, the broker is able to provide the messages to the client acting as a subscriber. In fact, this publish/subscribe queue delivers a copy of the topic related messages to each output queue. This also means that there can be more than one client that consumes the topic-specific messages and thus there can be more than one subscriber. Especially this feature of having multiple consumers of a message channel, is decisive for using a message broker in an environment where a distributed system processes the same stream of data in multiple ways (see chapter 5).[22] [31]

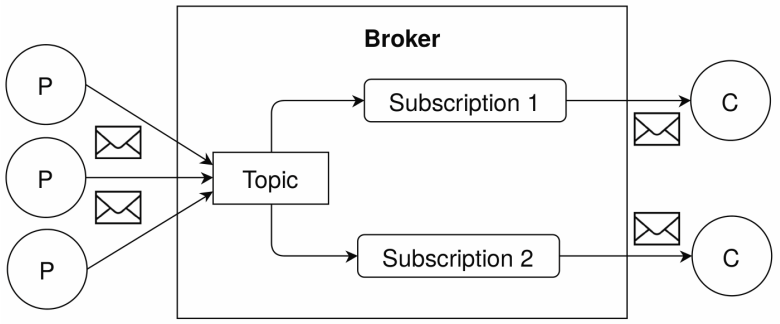


Figure 4.8.: Broker forwards published messages to subscribers

4.4.4. Reliable Delivery

A reliable delivery of messages can promise a specific behaviour to producers and consumers. A message broker typically supports one ore more of the following guarantees. *At most once* guarantees that messages may be lost but are never redelivered. *At least once* defines that messages are never lost but may be redelivered. The *Exactly once* semantic which is very typical guarantees that each message is delivered only once. Broker system which use the *In Order* semantic deliver messages to the consumer in the strict same order in which they are sent from the producer to the broker.

4.5. Characteristics of a Messaging System

As described, a messaging system has its strengths in interoperability and loose coupling of distributed computers. A message broker in turn has also many further characteristics. The ones most related to our work are:

Independence
: The more independent to any technology a message broker is, the better it can be integrated in a existing environment. The use of standards and effective standardized interfaces promotes independence.

Scalability
: Ability to dynamically increase performance depending on work load. Instead of scaling vertically which means to add resources to a single broker in the system, scaling horizontally is much more sustainable. The latter means to add more broker nodes to the system and evenly spread the load to all provided nodes. Because of synchronization, it is much harder to provided a broker system which can be scaled out horizontally but also much more efficient.

Latency
: Time it takes to process a single message for consumption.

Fault Tolerance
: Ability to recover after a failure with minimal message loss. Possibility to build redundancy through clustering and replication strategies such as master-slave or state machine replication. It is a significant characteristic how a message broker syncs its replications and elects which of the nodes to get active if the master fails. Also it is important that in case of failure the system can recover as quickly as possible and with minimal throughput deficit.

Persistency
: Ability to offer durable and persistent messages even if the broker systems restarts, crashes or consumer is inactive for a longer period of time (for instance batch processing systems).

Throughput
: Proportion of messages which can be sent to a broker to the amount of messages which can be consumed in a fixed time interval.

5. Introduction to Event Streaming

In previous chapter we defined what a message broker is, one can now imagine what systems like Apache Kafka basically intend. Because Apache Kafka is built for big data environments, concepts like events and streams become also important. To understand the possible use cases of Apache Kafka, the terms around event streaming need to be clarified.

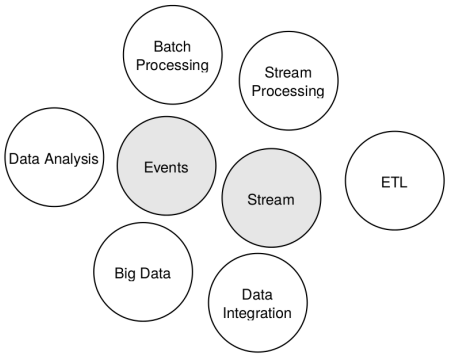


Figure 5.1.: Event Streaming terms

5.1. Purpose

The view of data as rows of databases or single files changes when one thinks about what a business actually does with the data generated in large distributed systems. Where retail generates orders that lead to sales, shipments and so on, a financial institution will generate orders that are going to have an impact on a stock price. Or a social network platform generates clicks, impressions and searches that are used to make some sort of intelligent analysis to further display personalized data to it's users. Such kinds of data can be thought of as streams of events. In fact, collecting all events a business ever generated will lead to the current state of the business and thus describe what the business did in the past. For example the current price of a stock was generated by all the orders ever made on this stock. Every order can be captured as an event and so can all events together can reproduce the current stock price.

5.2. What is an Event?

Very basically an event occurs when "something happens" in a system like when a user of an online shop adds an item to his basket. In modern systems, events are transmitted as discrete messages on a MOM (4.2) and thus following Tannenbaum et al. (2006) represent a data unit of a data streams. Where a data stream can be applied to discrete as well as

continuous media, events are transmitted as discrete messages only. The message itself can be considered as an event message. [22]

If we think more traditionally, even database systems can be thought of as event based systems. The process of creating a backup in form of dumps will not scale as we increase the frequency of dumps over time. Not only will the process take longer depending on the size of the database, also system resources are limited during this process. An approach to make this more efficient is change capture, which describes the difference between the state of the affected database rows before and after the change as an event. If this can be done continuously, a sequence of row changes is what is being left. This in fact, can be described as a stream of events.

Systems which interact with event basically have the following key objectives:

- **Data Analytics**:
  Preparing collected data for further analysis at any time.
- **Data Integration**:
  Making all the generated data of an organization available in all its services and systems.

5.3. Processing of Events (Data Analytics)

So data streams consisting of events by themselves are not valuable but can be taken advantage of by a system that processes these events, produces a result and provides it to other services. This can be the calculation of the new stock price after a customer sold his stock or the personalized content on a news feed after a person subscribed to a new fan page. But it could also be a more complex analysis over all the collected events that ever happened, stored in a big database.

In fact, the above mentioned examples differ in their nature. Where the calculation of the stock price is fairly simple by setting the price to the latest paid without any knowledge about the stock prices in past. A complete analysis of a huge data base will not only require a significant amount of processing time, it also requires some data produced in the past. This leads to two different approaches to handle an incoming event stream of any size:

Store raw data
: Simply storing every single event in a big data store. Through appending every incoming event one gets a detailed history of every activity on the system. To analyze the data, a periodic batch process can execute big queries over all events to get a result. ⇒ **Batch Processing**

Store aggregated data
: Instead of persisting every single event, directly process the incoming data stream and store only an aggregated summary. Due to the updating of the aggregation with every incoming event, getting an overall result is very fast (which we call "Real-Time"). ⇒ **Stream Processing**

[24]

5.3.1. Batch Processing

Traditional batch processing systems nowadays are distinguished between map-reduce-based and non-map-reduce-based systems.

The process of data integration (a.k.a data extraction-transformation-load, ETL), runs at a regular time interval, such as daily, weekly or monthly. Analyzing data that resides in a data store stage becomes challenging when data size grows and systems may not be able to process results within a reasonable time limit. [26]

As the trend shows, the needs of performance and responsiveness in a big data environment can't be fulfilled with traditional batch processing anymore. Instead, real-time processing becomes more important than ever to achieve results from queries in minutes, even seconds. [19]

In real-time batch processing fashion, systems will address the data integration stage with continuous input of data. Processing in near-real-time to present results within seconds is being addressed in data analytics. Thus, real-time batch processing gives organization the ability to take immediate action for those times when acting within seconds or minutes is significant. [23]

5.3.2. Stream Processing

Stream processing refers to integration and processing of data before storing. A stream processing system is built out of multiple units called processing elements (PE). Each PE receives input from its input queue, does some computation on the input using its local state and produces output to its output queue. PEs communicate always through messaging with other PEs.

Most importantly, those systems are optimized for low latency and high availability. Recovering from failures is critical for stream processing systems and should be fast and efficient. Data should be partitioned and handled in parallel for large volumes of data. The partitioning strategy of a system affects how the system handles the data in parallel and how the system can scale. [23]

Stream processing frameworks – such as Storm, Samza, or Spark Streaming – were specially developed to provide rich processing primitives and thus can be taken advantage of in the data integration- and processing stages.

5.3.3. Lambda Architecture

While batch processing is used for data analysis to get results out of huge amounts of stored raw data at any time, stream processing reacts to events in real time. Both approaches are very useful in different use cases. Lambda architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream processing methods. It is split into three layers, the batch layer, the serving layer and the speed layer:

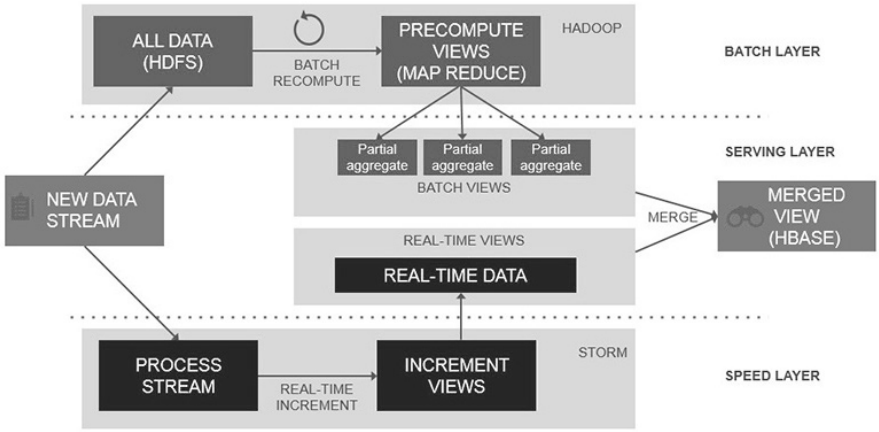


Figure 5.2.: Lambda Architecture [17]

Any query is answered through the serving layer by querying both the speed and the batch layer. Where the batch layer periodically computes views on the current collected data and is being outdated at the end of its computation, the speed layer closes this gap by constantly processing the most recent data in near real-time fashion. [28] [23]

5.3.4. Requirements to Event Processing Systems

Any system which is dependent on a continuous input of event data, requires a delivery system that can provide data constantly as a stream. Stream processing systems, whether being in a lambda architecture or not, obviously fulfill this requirement. On the other hand, in a big data environment there is also the requirement of batch processing systems (5.3.1) being served with continuous data. In a lambda architecture this could be done using a stream processing framework (5.3.2) responsible for serving the batch processing system with integrated data, ready for data analysis. However, an other way of doing so would be a data store – such as the hadoop file system (HDFS) – where data can be directly taken for further analysis. The same requirement of a data store holds for any other business intelligence system followed by the problem of depending on a data store which eventually – due to the lack of an adapter – can not be served with data by a stream processing system as comfortable as the HDFS.

Facing the two types of processing events – stream- and batch processing – together, data integration results as a common stage both systems have to deal with. Even further, for every system an organization operates, like the data warehouse, full-text search indexes, caches or more, the data integration stage will be present. Additionally, stream processing systems require a continuous incoming stream to further integrate and process where batch processing systems on the other hand demand a given persistent set of data to further integrate and analyze.

Further more, in terms of stream processing the requirement of low latency is essential for any system of this type. For database systems combined with stream processing, reliability becomes significantly important to handle critical updates such as replicating as discussed above. In terms of batch processing however, the demand on low latency is not as important as the availability of well integrated data with a high throughput to be able to handle a large volume of data in time range as low as possible.

	Stream Processing	Batch Processing
Data Integration	x	x
Continuous Data	x	
Persistent Data		x
Low Latency	x	
High Throughput		x

Table 5.1.: Requirements of Batch and Stream Processing Systems

5.4. Event Stream (Data Integration)

The need of a constant input of event data is discussed as general requirement for event processing (see table 5.1). A technology or system which provides this constant flow of events from any system to event processing, is part of the data integration and is called event stream in a highly abstract manner.

5.4.1. Mesh of Streams

Note that the mentioned event processing systems do not only consume data for integration and processing, they can also have an output. Thus, many systems are both sources and destinations for data transfer in consequence of which a system would then need two channels per system. Obviously, each output in this constellation is intended to be consumed by 1–N system(s) again. Connecting all of these would lead to building a custom channel between each pair of system (see Figure 5.3). As the set of systems in an organization grows, this would clearly become a challenge to maintain.

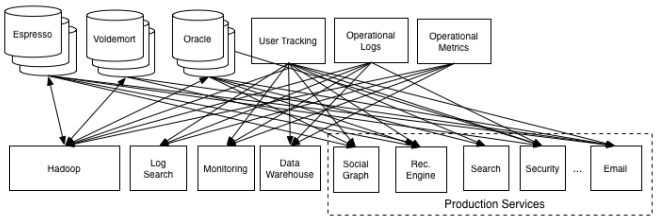


Figure 5.3.: Complex Point-To-Point Architecture

In this scenario several hurdles arise. The process of data integration would have to be done for each system pair individually. At this point, an obvious approach for a simplification would be to introduce an organization-wide standardization of the data format. Thus, the integration process becomes significantly easier but still has to be done redundantly. Another problem that remains are the tightly coupled systems. A simple change on one system could affect one or more of its connected systems directly, which is not only hard to manage but also reduces the flexibility of further development of the landscape. To extend the landscape the chances are high to touch existing systems which is not only time intensive but also connected with risks regarding possible failures. In fact, these are known issues of a Point-To-Point channel of traditional messaging and are described in Section 4.3.1.

5.4.2. Central Platform as Solution

A more elegant and reliable approach to solve the mentioned problems would be to introduce a central platform, that is able to support both batch and real-time consumption and thus is able to fulfill the described requirements given in table 5.1. This system can act as a single data repository to isolated consumers and gives access to any data that is required, as shown in figure 5.4. It aggregates incoming events and represents an even stream for any its consumers.

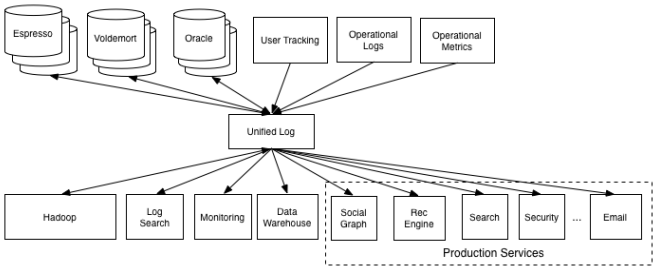


Figure 5.4.: Architecture with a centralized event stream platform

In this constellation integrating a new data system – be it a data source or a data destination – is fairly simple. Connecting it to a single channel attached to the central platform is all that needs to be done. Besides the loosely coupled components, with this architecture it becomes possible to centralize the process of data integration by doing so in a standardized way, directly within the stream. Thus, a huge factor of complexity over the system landscape is being reduced. In fact, this opens up a whole new set of possibilities in organizational scalability.

5.5. Link to Message Brokers

As solution for a modern big data environment with a huge amount of event data which needs to be processed in different systems we need a platform which acts as a mediator to handle the incoming events as streams and provide them to several consumers (data integration). This seems to be very similar to the definition of a message broker (4.3.2), and indeed these two concepts can be compared. Actually, an event stream can be realised with an underlying message broker. Events can be considered as messages with the event data as payload. The central stream platform is nothing else than a message broker which handles incoming events, persists them in an ordered queue and provides them for the consumers which can be stream or batch processing systems.

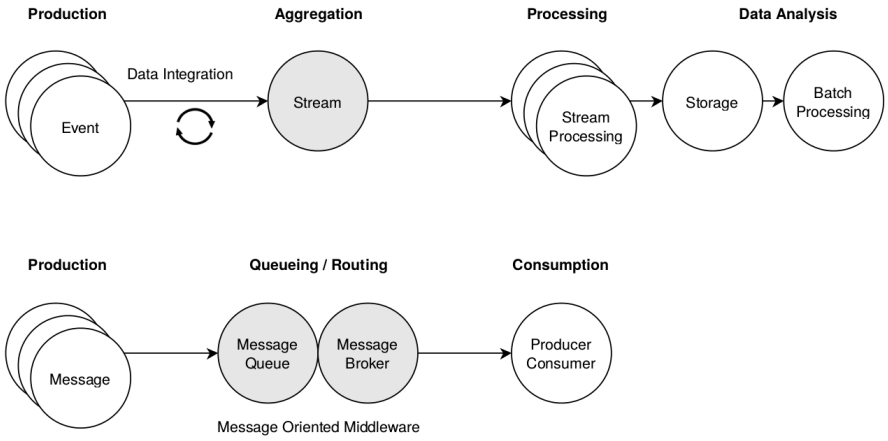


Figure 5.5.: Event Stream vs Messaging

The question if a message broker implementation is predestined for a big data event stream environments is depending on its guarantees and performance. In comparison to a standard integration platform where a few applications communicate with each other, the main challenge for a message broker in a big data environment, is the very large amount of data which needs to handled. The main demands are high throughput and reliability in case of failures.

5.5.1. Example Implementations

The following implementations give an overview of existing message broker systems. In chapter 6 we will then examine the capabilities of Apache Kafka related to the functionalities of acting as a centralized event stream platform.

Rabbit MQ

RabbitMQ is written in Erlang and its source code is released under the Mozilla Public License. It supports many messaging protocols, among which the most important are STOMP: Streaming Text Oriented Messaging Protocol and AMQP: Advanced Messaging Queuing Protocol. Incoming messages are placed in a queue whereas this message can further be stored in memory or on a disk. The latter model of persistency will under-perform when there is a large number of queues which need to access the disk simultaneously. [18]

In a RabbitMQ Cluster, queues are created and live in a single node, and all nodes know about all the queues. When a node receives a request to a queue that is not available in the current node, it routes the request to the node that has the queue. To provide high availability (HA), RabbitMQ replicates messages between master and a mirrored slave, so the slave can take over if the master has died. [34]

What RabbitMQ clustering does not providing guarantees against message loss. When a Rabbit cluster node dies, the messages in queues on that node can disappear. This is because RabbitMQ doesn't replicate the contents of queues throughout the cluster. They live only on the node that owns the queue. [32]

Active MQ

Apache ActiveMQ is an open source message broker written in Java together with a full Java Message Service (JMS) client.

The messages are stored in a data log (called journal), which consists of individual files. When all messages in a data log have been successfully consumed, the data log file is marked as being ready to be deleted - or archived in a periodical cleanup cycle. Then at intervals the journal can be copied to a long term persistence storage (e.g. JDBC). Multiple brokers nodes cannot share a journal. Each must be configured with it's own journal. The journal cannot be used, however, to recover messages as it does not keep an ordered index. So the long term store is used to recover the durable subscription. Additionally a separate in-memory reference store holds references from those messages residing in the data log to improve performance.

In a clustered long term persistence setup, Apache Zookeeper is used to pick a master from a set of broker nodes configured to replicate a LevelDB Store. Then synchronizes all slave LevelDB Stores with the master keeps them up to date by replicating all updates from the master.

ActiveMQ will preserve the order of messages sent by a single producer to all consumers on a topic. If there is a single consumer on a queue then the order of messages sent by a single producer will be preserved as well. However due to multi-threading and asynchronous processing, the messages from different producers could arrive in different consumers in different orders. [1]

Amazon SQS

Amazon Simple Queuing Service is a cloud service which offers a simple message queue to move data between distributed systems. SQS guarantees reliability by redelivering Messages in case of failure. If a message is successfully processed by a consumer it will be deleted out of the queue. Amazon SQS does not guarantee a strict ordering of messages. All messages are stored redundantly on multiple servers and in multiple data center. [5] [2]

SQS as itself can not provide a publish subscribe behaviour as it is available in typical message brokers. But in combination with the Amazon Simple Notification Service (SNS) it is possible to create topics which are linked to SQS queues where consumers can register itself and only receive message from a specific topic. [15]

Amazon Kinesis

Amazon Kinesis is a cloud service for streaming event data for further processing which is very similar to Apache Kafka. Producer applications push data continuously to the Kinesis broker where the messages are kept for 24 hours in memory. Kinesis provides ordering of records, as well as the ability to read and/or replay records in the same order to multiple Amazon Kinesis Applications. [4] [3]

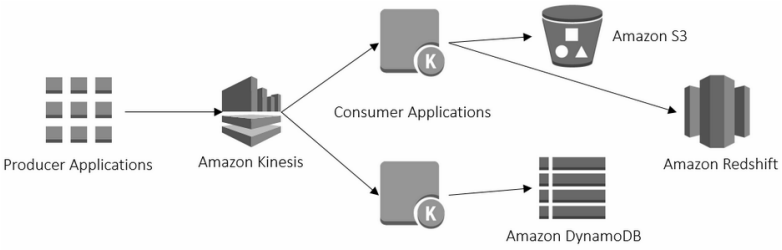


Figure 5.6.: Amazon Kinesis aggregates events and interacts with other Amazon Cloud services [4]

Apache Flume

Flume is a distributed service specialized on being a reliable way of getting event data into HDFS. The typical deployment consists of a number of logical nodes, arranged into three tiers. The first tier is the agent tier. Agent nodes are typically installed on the machines that generate the data and are the initial point of contact with Flume. They forward data to the next tier of collector nodes, which aggregate the separate data flows and forward them to the final storage tier. Flume is push based and does not support public-subscribe semantics. [6]

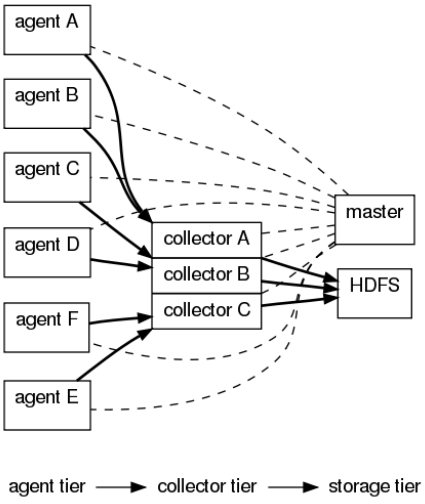


Figure 5.7.: Apache Flume general architecture [6]

6. Apache Kafka

As this thesis is about the implementation of a message broker that correlates to the functionalities Apache Kafka provides, we describe in the following paragraphs the components and functionalities of Apache Kafka in its current state (Version 0.8.2). We will not provide any implementation-specific details in this section but describe the concepts in a higher level of abstraction instead. Thus, it is also an overview and the initial position of our own implementation.

6.1. Background

Apache Kafka was initially developed at LinkedIn[16] and subsequently released as an open source project with the Apache Software Foundation[7].

Initially at LinkedIn the landscape was overwhelmed by the complexity due to point-to-point (4.3.1) pipelines that delivered data to a single destination with no integration in between. Messages were written to aggregated files and then copied to ETL servers (data integration) and further loaded into data warehousing and batch processing (5.3.1) clusters. Thus, this service suffered the lack of real-time data access which is – especially for a data driven company that populates activity-driven news feeds – an essential requirement. In fact, this fragility and complexity of this pipeline lead to inevitable delays in adding new types of activity data. Besides the feature-wise limitations, also the detection time of operational problems increased over time due to the ever-increasing pressure on the latency of the data warehouse processes. This could have been solved with stream processing systems (5.3.2) but was not supported at this point, due to the lack of any central platform (5.4.2) that can provide continuous data. [21]

First attempts towards a piece of infrastructure (e.g. broker) that can serve stream and batch processing systems were made by experimenting with ActiveMQ[1]. During tests under full production load they ran into several significant problems. It turned out that if the queue backed up beyond what could be kept in memory, performance would be severely degraded due to heavy amounts of random I/O. Inefficiencies had to be accepted regarding clustered consumers requiring duplicating the data for each consumer in a separate queue. Further difficulties were faced with ActiveMQ's built in persistence mechanism that lead to very long restart times.

According to LinkedIn it would have been possible to provide enough buffer to keep the ActiveMQ brokers above water but would have required hundreds of servers to process a subset of activity data. As a result, the decision was made to build a custom piece of messaging infrastructure targeting high-volume scale-out deployment and thus serve batch and stream processing systems. [21]

6.2. The Log

6.2.1. What is a Log?

Apache Kafka uses a so-called commit log for managing messages within the broker. This log has nothing in common with "application logging", as traces of a software designed for human reading. Instead, logs in the context of distributed systems are designed for programmatic access. Basically it is a simple, append-only data structure which contains a sequence of records ordered by time where each entry is assigned to a unique number called offset. Thanks to the strict ordering inside a log the record offset can be used as a timestamp where a log gets decoupled from any time system. [8] [25]

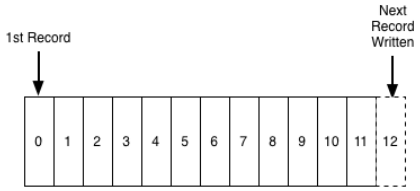


Figure 6.1.: The Log [25]

Apache Kafka handles its own log for every topic which contains all published messages as single records. Compared to traditional messaging systems, Kafka does not delete a record after consumption, actually it makes no difference whether or not they have been consumed. As we see later, every consumer controls its position of the log on its own. Kafka can hold records within a defined time window before it deletes the oldest records of the log. Further more, an additional feature called log compaction can be activated to reduce the amount of messages which need to be deleted by removing only obsolete records. [8] [25]

Logs originate from databases where they are used for replication and recovery. Every change or update of a log leads to a new state that represents a version including all the updates done in past. By describing every replica by the latest log entry it has processed, the problem of coordinating states of distributed replicas is getting much easier. Apache Kafka uses this approach for consumption of messages. Every consumer knows its current offset and increases it linearly as it reads messages from the log. The log can be seen as a re-playable record of history whereas a consumer also can reset to an older state – by using the offset – to reprocess. [25]

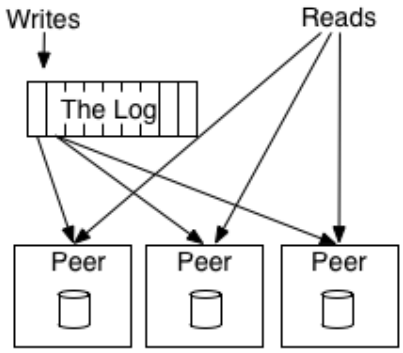


Figure 6.2.: Peers can sync their state with the log [25]

6.2.2. Persistence

Apache Kafka writes all data immediately to a persistent log (6.2.1) and therefore relies heavily on the file system. Instead of flushing incoming messages from producers directly to the disks, the data is stored as a compact byte structure within the memory.

The process of copying the data to the disk is handled by the operating system that will not only use linear reads and writes as a pattern for heavy optimization but also provides read-ahead and write-behind techniques. The latter will pre-fetch data in large block multiples and group smaller logical writes into large physical writes. Thus, a SATA RAID-5 of six 7200rpm disks will achieve a performance of 600MB/s in writes.

As a result of of the mentioned factors of using the file system leads to the fact that Apache Kafka claims access to all free memory. Doing so will result in a cache of up to 28-30GB on a 32GB machine. In a default setup, data is kept in memory for seven days to provide data required for processing directly from the memory. This duration should be set according to one's own needs. In fact, the memory stays persistent during a restart of the Kafka service but won't if the server appliance reboots. Then the operating systems flushes the memory. In the latter scenario, Apache Kafka will restore all data from the hard disk to the memory. [8]

One drawback following this model of not immediately writing data to disk can not be omitted. A small number of messages can be lost in the event of a hard server failure before messages are transfered to the Kafka brokers or flushed to disk. [21]

6.2.3. Partitioning

Facing the challenges of large distributed systems, a log must scale well. For improving scalability and fault tolerance, Kafka can divide a log in to multiple partitions. This is a way to parallelize the consumption of the messages, whereas the total number of partitions in a broker cluster needs to be at least the same as the number of consumers (or producers) in a consumer (or producer) group. Each partition is consumed by exactly one consumer (in a consumer group) and by doing so, Kafka ensures that the consumer is the only reader of that partition and consumes the data in order.

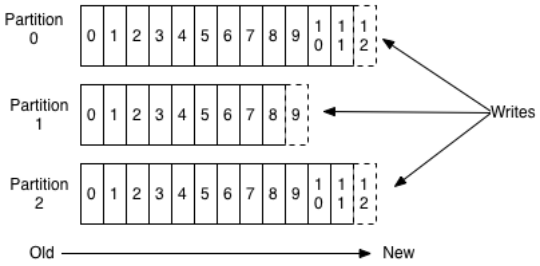


Figure 6.3.: Partitioned Log [8]

Each member – be it on consumer or producer side – will split the burden of processing the topic between themselves according to the partitioning so that one member will only be concerned with messages in the partition it is assigned to. Thus, the throughput of the system will scale linearly with the Kafka cluster size. [8]

6.3. Replication

To hold the defined guarantees (6.5) regarding to system or network failures, Apache Kafka supports replication on the level of log partitions (6.2.3). Thereby it can replicate specific topics across a configurable number of other Kafka servers (a.k.a. nodes). Every topic has defined a leader node which is active in normal operation. Producers send data directly to the leader broker. A leader node can have zero or more follower nodes which are responsible for replicating the entries of the active log. The followers do this by simply acting as a normal Kafka consumer of the leader node and constantly update their own log so that it is identical. [8]

If a leader node crashes, Kafka needs to choose a new one from among the followers. Instead of using the common algorithm of quorum replication, it has its own approach called in-sync replicas which is based on primary-backup replication. In this algorithm the leader node constantly keeps track of its followers by managing a set of *in-sync* nodes, called ISR (in-sync replicas). If one of these nodes crashes or falls behind in replicating, the leader will dynamically remove it from its ISR. If a follower comes back and catches up to the actual state the leader will reinsert it. Only members of this set can be elected as the new leader. An incoming message needs to be replicated by every *in-sync* follower before any other consumer can get it. A fully replicated message is considered as *committed*. This guarantees that the consumer does not need to worry about potentially seeing a message that could be lost if the leader fails. [8] [10]

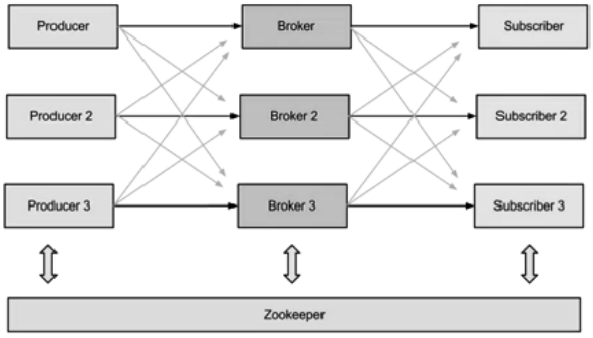


Figure 6.4.: Zookeeper as underlying clustering manager [9]

Apache Kafka always runs with Apache Zookeeper as the underlying manager for all Kafka nodes. It handles the coordination between the leader and follower nodes and also notifies producer and consumer about the presence or failure of brokers in a Kafka system. The latter leads producers and consumers to coordinate their work with different brokers. Regarding the replication of Kafka nodes, Zookeeper also handles the leader election in case of failure. In order to do this, a leader node constantly informs Zookeeper about changes in his ISR set. If the leader node crashes, all surviving followers register themselves in Zookeeper. The replica that registers first becomes the new leader. [10] [9] [11]

6.4. Clients

6.4.1. Producer

A producer can publish data to a topic of its choice and is responsible for choosing which message will be assigned to which partition within a topic. In fact, the producer sends data directly to the leader of a partition on the broker. In order to help the producer find the correct server and partition leader, Kafka can answer a request for meta-data about which servers are alive and where the leaders for the partitions of a topic are at any given time. Thus the producer can direct its requests appropriately. [8]

The Kafka producer client can be configured to send messages in either a synchronous or asynchronous fashion. The asynchronous mode allows the client to batch (6.4.3) small messages into larger data chunks before sending them over the network. This can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (e.g. 64 kilobytes or 10 milliseconds). Thus, batching allows fewer but larger I/O operations on the servers and results in higher throughput. [8] [21]

6.4.2. Consumer

Apache Kafka introduces the concept of *consumer groups* where consumers label themselves with a consumer group name, and each message is delivered to one consumer instance within each subscribing consumer group. Consumer groups can be in separate

processes or on separate machines.

In fact, this concept generalizes traditional queuing (4.2) and publish-subscribe models (4.4.3). If all consumer instances have the same consumer group, then it works just like a traditional queue balancing over the consumers. On the other hand, if all the consumer instances have different consumer groups, it works like publish-subscribe.

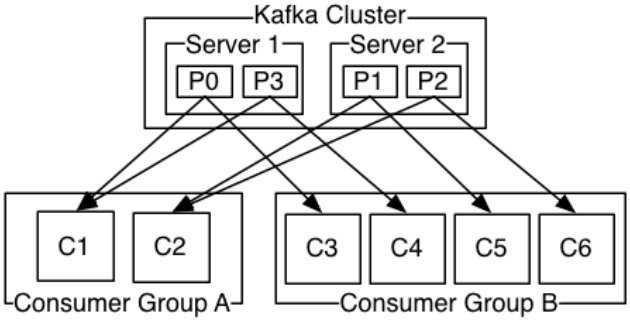


Figure 6.5.: A two-server Kafka cluster hosting four partitions with two consumer groups. Consumer group A has two instances and group B has four [8]

A consumer specifies its offset in the log within each request sent to the broker and in return will receive back a chunk of log beginning from that position. This leads to the fact that Kafka relies on the pull-model (as described in 4.5) in order to reach a maximum performance on the consumption side. As a side effect, the broker will be freed from additional complexity regarding consumer handling from the broker. Another advantage of Kafka relying on the pull-model results in the possibility of aggressive batching of data sent to the consumer (see 6.4.3). Apache Kafka avoids the naive approach of busy-waiting for data to arrive but rather allows the consumer request to block in a "long poll". [8]

Unlike most other messaging systems, Apache Kafka does not keep track about what messages have been consumed on the broker and thus does not rely on any acknowledgements from the consumer. Instead, the position of a consumer is just a single integer on a partition which defines the offset of the next message to consume (see also 6.2.1) for each specific consumer. As a side benefit, a consumer can rewind back to an old offset and re-consume data. The idea is to minimise the amount of state that Apache has to keep. Replicated state is what makes fault-tolerance hard. [8]

6.4.3. Batching

The trade-off between latency and throughput is addressed in Apache Kafka by grouping small messages together – referred to as batching. Rather than relying on Nagle's algorithm on a TCP level, batching on the producer side is implemented at the application level which allows precise timeout and message count thresholds that trigger the sending. The broker itself does not do further batching of file system writes but rather relies fully on the file system (as described in 6.2.2) which brings an improvement so significant that it would not make sense to further try to improve manually. Consumer data fetch requests are also batched effectively by allowing the user to pull data from many topics and partitions in a single request to avoid small requests on low-volume topics. [21]

6.5. Guarantees

As a resume from the descriptions above we list guarantees of Kafka regarding message delivery, fault tolerance and ordering.

Delivery Model

Kafka guarantees at-least-once delivery by default and allows the user to implement at-most-once delivery by disabling retries on the producer and committing its offset prior to processing a batch of messages. Exactly-once delivery requires co-operation with the destination storage system but Kafka provides the offset which makes implementing this straight-forward.[8]

Fault Tolerance

It is guaranteed that any successfully published message will not be lost and can be consumed. Further more, a topic with replication (6.3) factor N will tolerate N-1 server failures without losing any messages. However there is no guarantee for the producer that a message is fully published in case of a network error. A feature which enables the producer to retry publishing a message as an idempotent request may be added in a later version of Kafka. [8]

Message Ordering

Thanks to the use of a log with sequential numbers for each entry (offset), Apache Kafka guarantees that messages from a producer to a topic partition will be appended in the order they are sent and consumers see messages in the order they are stored in the log. [8]

Part III.

Technical Report

7. Specification

7.1. Purpose

The technical research of this thesis gives an insight into the fundamentals of messaging, especially the purpose of message brokers for big data event streaming. Also detailed information about Apache Kafka are provided. The second part of this work is focusing on the implementation of a message broker in the functional programming language Haskell[1]. The goal is to make use of the advantages of the functional paradigm regarding the resulting throughput performance. By using the example of Apache Kafka, the main purpose lies on building a server application which implements basic functionality of producing and consuming messages.

The implemented broker application is named as Haskell Message Broker whereas we will use the short form HMB by now.

[1] https://www.haskell.org/

7.2. Components

Basically, the architecture consists of two main components, namely the broker and the client, whereas the client can act as producer, consumer or both. The broker is a server which only reacts to requests that are sent from clients. Every request contains an API key which the broker uses to determine which action it has to do (e.g. persist produced message or consume message). For each valid request the broker sends back a corresponding response to the client which either includes the fetched data or an error code. The broker never communicates with a client without a request.

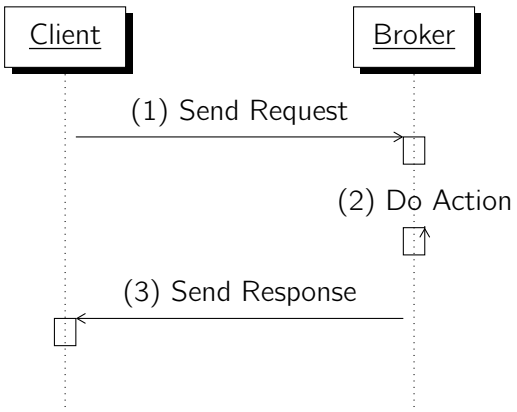


Figure 7.1.: Basic communication between client (producer or consumer) and broker

Both, the client and broker component need to use the same underlying protocol for communication. Because Apache Kafka is used as the benchmark for this project and compatibility to existing Kafka clients is to be provided, the Kafka Protocol version 0.8.x has been chosen to be implemented. Therefore, a third component which fully implements this protocol is introduced additionally. It exposes the appropriate functions and types as a library. This component fully separates the clients from the broker.

To support interoperability, the implementation of a client should be simplified as much as possible. Therefore we provide a client library which exposes functionalities to easily implement a Haskell client for a Kafka-based broker. The goal of this, is to be able to set up a client without knowing too many details about the Apache Kafka protocol itself.

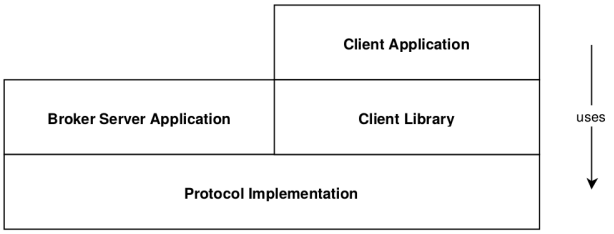


Figure 7.2.: Seperation to four components

7.3. Workflow

The main purpose of this broker implementation can be split in two cases. Case one covers producing a message and persisting in the broker's log. Case two allows to consume persisted messages on request. As for clients, depending on which API is to be used, the client can act either as producer or consumer.

7.3.1. Produce Messages

In the case of a producer client, the following workflow highlights the fundamental steps being passed from sending a request by the client to persisting the message on the broker.

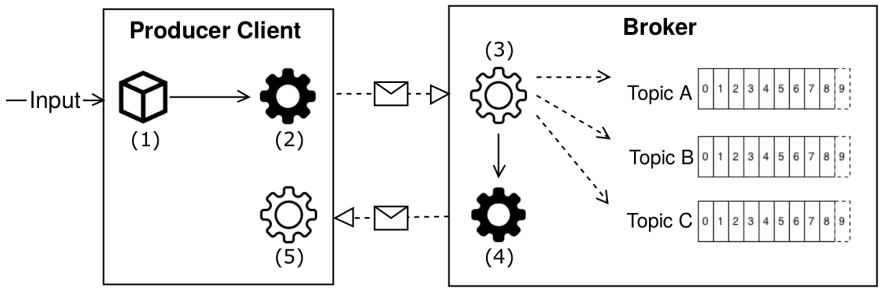


Figure 7.3.: Case one: Producer Workflow

(1) Packing Produce Request: Transforming input data into protocol-conforming data structure.

(2) Serializing and sending produce request: Encoding data structure to a binary data and transmit over network to the broker.

(3) Parsing and handling produce request: Broker receives the binary string and parses it back to the appropriate data structure. The request handler of the broker checks the API Key of the request. If it is a produce request, the containing message will be written to the appropriate topic log.

(4) Send produce response: A response is packed, serialized and transmitted back to the client. The response contains an error code which determines if everything worked well or a specific problem occurred.

(5) Parse produce response: Producer client receives a binary string and parses it to a valid response data structure

7.3.2. Consume Messages

In the case where the client acts as a consumer, the workflow is not different from the one for producing a message. Instead of the translation from request message to the log, persisted messages are read from the log and are delivered from the broker to the client.

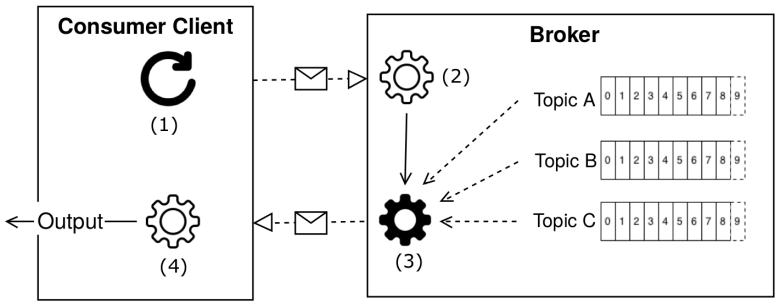


Figure 7.4.: Case two: Consumer Workflow

(1) Continuously send fetch request: Consumer client sends fetch requests in configurable interval as a binary string to the broker.

(2) Parsing and handling fetch request: Broker receives the binary string and parses it back to the appropriate data structure. The request handler of the broker checks the API Key of the request. If it is a fetch request, the broker reads messages of the requested topic and packs them to a fetch response.

(3) Send fetch response: The fetch request which contains the requested messages is sent back to the consumer client.

(4) Parse response: Consumer client receives a binary string and parses it to a valid response data structure.

8. Implementation Protocol

This chapter provides details of the implementation of the Kafka protocol, which acts as a basis for all further functionality. The definition of the original protocol is given as a context free grammar for the request and response binary format [14]. Therefore it seemed appropriate to use the rules of the grammar to our protocol implementation by mapping them to the Haskell type system. In a second step encoding and decoding functionality was implemented for getting messages from data structures to binary formats and backwards. In addition to the protocol, a client library will be provided to isolate client implementations from protocol implementation but also allow to build simple Apache Kafka or HMB clients in Haskell without touching the protocol at all.

The protocol implementation consists of the following modules:

- Types: Mapping the protocol definition to the Haskell type system.
- Decode: Provide functions to parse binary data to valid structures.
- Encode: Provide functions to serialize a given structure to a binary format.
- Client: Provide functions to simplify a Haskell client.

8.1. Types

The design of the Apache Kafka protocol allows to make a distinction between three kind of types:

1. Related to request
2. Related to response
3. Related to data (for either request or response but can also be used for the Apache Kafka Log[1] component since log files (on-disk) hold the same structure)

As mentioned before, the representation of the protocol structure is implemented using the Haskell type system[2], more specifically using data types created for our needs using named fields - also known as record syntax[3]. After this, a further step of abstraction is introduced by creating types for the binary representation (8.1.3) of a protocol field.

8.1.1. Naming Convention

For request – and response – related types a custom naming convention is introduced as it is a fact, that not every field of some kind of request or response will be unique. There are several fields that will represent a *topicName* for example. Thus, naming the field *topicName* would have been the obvious solution but since the record syntax in Haskell won't

[1] http://kafka.apache.org/documentation.html#log
[2] https://wiki.haskell.org/Type
[3] http://en.wikibooks.org/wiki/Haskell/More_on_datatypes

allow to use the same name for a field twice - even in different data types - unique prefixes were defined for each request and response. The following table provides information about the types related to request or response as well its prefix:

API	**Request (Rq)**	**Response (Rs)**
Metadata API (Md)	MetadataRequest	MetadataResponse
Produce API (Pr)	ProduceRequest	ProduceResponse
Fetch API (Ft)	FetchRequest	FetchResponse
Offset API (Of)	OffsetRequest	OffsetResponse
Offset Commit API (Ofc)	OffsetCommitRequest	OffsetCommitResponse
Offset Fetch API (Oft)	OffsetFetchRequest	OffsetFetchResponse

Table 8.1.: Naming Convention Protocol Types

As a result, a typical data type is described as follows, while details are hidden for demonstration purposes.:

```
1  -- more
2  data Partition =
3      RqPrPartition
4      { rqPrPartitionNumber :: !PartitionNumber
5      , rqPrMessageSetSize  :: !MessageSetSize
6      , rqPrMessageSet      :: [MessageSet]
7      }
8      |
9      RqFtPartition
10     { rqFtPartitionNumber :: !PartitionNumber
11     , rqFtFetchOffset     :: !Offset
12     , rqFtMaxBytes        :: !MaxBytes
13     }
14     |
15 -- more
```

Listing 8.1: Example of defined protocol types with name convention

8.1.2. Batching

A significant characteristic of the Kafka protocol is the ability to optimize efficiency due to the batching of multiple messages in one single request. Both the API to send messages and the API to consume messages always work with a sequence of messages and not only a single message. It is also possible to batch messages across multiple topics and partitions. It is the task of the client implementation to use this ability cleverly but it also can ignore it and sends everything one at a time. Batching leads to sequences of the same type in one request or response. Within the protocol format this is defined by an array type. In our protocol implementation the batching sequences are implemented with Haskell lists which contains N repetitions of a specific type. Sequences can also be nested.

For example, the following grammar rule of the original protocol definition shows the support of batching in the protocol. A batching sequence of a structure A is shown as [A]: [14]

```
ProduceRequest =>
    RequiredAcks Timeout [TopicName [Partition MessageSetSize MessageSet]]
```

8.1.3. Primitive Types

The actual type for the data fields in the protocol have to match an unsigned integer type of its length. This can be done using the Data.Word[4] library. However, while repetitively writing *WordX* (where X stands for 8, 16, 32, or 64 bit) is rather intuitive, creating aliases using the *type* keyword will result in better readable code, as well as structure of the implemented protocol.

Primitive Type	**Description**	**Mapping with Haskell library**
Fixed Width Primitives	Signed integers stored in big endian order.	Data.Word8, Word16, Word32, Word64
Variable Length Primitives	Consist of a signed integer with a given length N followed by N bytes of content. A length of -1 indicates null. String fields uses an int16 (Data.Word16) for its size, and byte-value fields uses an int32 (Data.Word32).	Data.ByteString
Arrays	Repeated structures are encoded as an int32 size containing the length N followed by N repetitions of the structure which again can contain primitive types	Data.List

Table 8.2.: Primitive types of protocol implementation

8.1.4. Error Codes

To handle errors, the protocol specifies error codes that can be sent within a response message. To adapt those codes, a data type `ErrorCode` deriving from **Enum**[5] is provided and implicitly represents the appropriate error code by its order.

```
1  data ErrorCode =
2          NoError
3        | Unknown
4        | OffsetOutOfRange
5        | InvalidMessage
6        | UnknownTopicOrPartition
7        | InvalidMessageSize
8        | LeaderNotAvailable
9        | NotLeaderForPartition
10       | RequestTimedOut
11       | BrokerNotAvailable
12       | ReplicaNotAvailable
13       | MessageSizeTooLarge
14       | StaleControllerEpochCode
15       | OffsetMetadataTooLargeCode
16       | OffsetsLoadInProgressCode
```

[4] http://hackage.haskell.org/package/base-4.7.0.2/docs/Data-Word.html
[5] http://hackage.haskell.org/package/base-4.8.0.0/docs/Prelude.html#t:Enum

```
17          | ConsumerCoordinatorNotAvailableCode
18          | NotCoordinatorForConsumerCodeA
19            deriving (Show, Enum)
```

Listing 8.2: Defined types for ErrorCode

8.1.5. Types related to Request

A request message is the on-wire format of data which is transmit from client to broker.

Each request message has the same header fields no matter for which API they are used. The real payload of the request is individual to the particular API which is used.

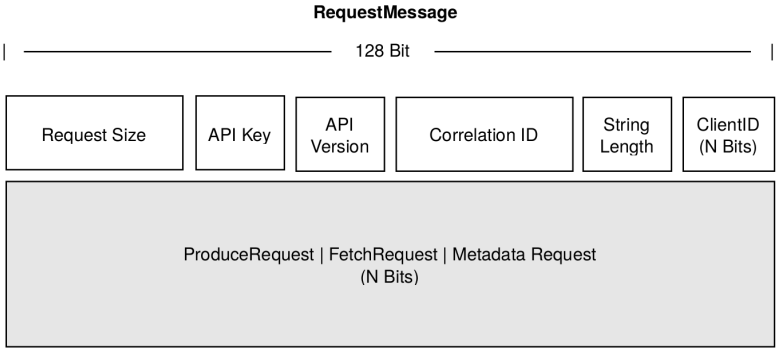


Figure 8.1.: Format of type RequestMessage

RequestSize	Data.Word32	Gives the size of the subsequent request message in bytes. The client can read requests by first reading this 4 byte size as an integer N and then reading and parsing the subsequent N bytes of the request.
ApiKey	Data.Word16	This is a numeric id for the API being invoked
ApiVersion	Data.Word16	Numeric version number for API (default value: 0)
CorrelationId	Data.Word32	Will be passed back unmodified in the response by the broker. It is useful for matching request and response between the client and server.
StringLength	Data.Word16	Length in bytes of the string ClientId
ClientID	Data.ByteString	Identifier for the client application
Request		API specific Request type (either ProduceRequest, FetchRequest, or others)

Regarding to the ApiKey the following numeric codes are already provided by the current protocol implementation (other API's are not implemented yet):

API Name	**ApiKey Value**
ProduceRequest	0
FetchRequest	1
MetadataRequest	3

Table 8.3.: Implemented API's with ApiKey Value

ProduceRequest

Request Payload for Produce API. Sends MessageSet's (see 8.1.7) to the broker. For efficiency it allows sending MessageSet's intended to batch multiple topic partitions in a single request (see 8.1.2).

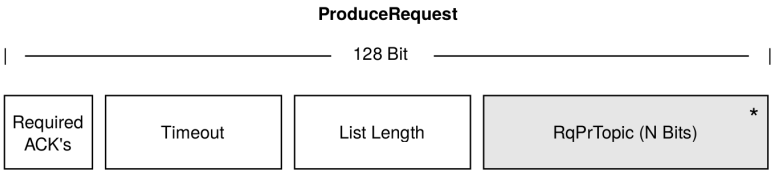


Figure 8.2.: Format of type ProduceRequest;

RequiredAcks	Data.Word16	This field indicates how many acknowledgements the broker should receive before responding to the request.
Timeout	Data.Word32	This provides a maximum time in milliseconds the broker can await the receipt of the number of acknowledgements in RequiredAcks.
ListLength	Data.Word32	Number of elements in the following list.
[RqPrTopic]	Data.List	Sequence of RqPrTopics, see below.

A ProduceRequest includes N repetitions of the RqPrTopic type, the producer client being able to send messages for multiple different topics:

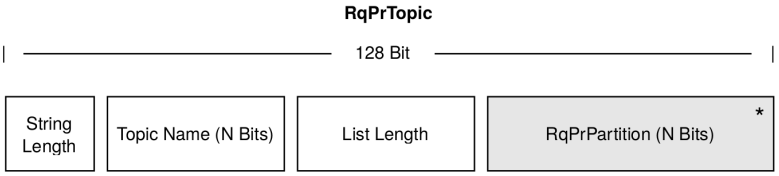


Figure 8.3.: Format of type RqPrTopic;

StringLength	Data.Word16	Length in bytes of the string TopicName.
TopicName	Data.ByteString	Name of the topic that data is being published to.
ListLength	Data.Word32	Number of elements in the following list.
[RqPrPartition]	Data.List	Sequence of PrPartitions, see below.

The RqPrTopic includes N repetitions of the type RqPrPartition so that the producer client being able to send message for multiple partitions within a topic.

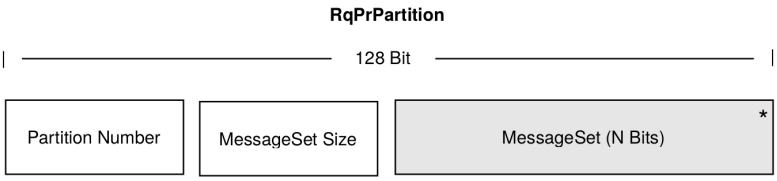


Figure 8.4.: Format of type RqPrTopic

PartitionNumber	Data.Word32	The identifier of partition where data is being published to.
MessageSetSize	Data.Word32	Sequences of MessageSet's are not preceded by an int32 like other array elements in the protocol. Instead, this field determines the total size, in bytes, of the following MessageSet's.
[MessageSet]	Data.List	Finally the RqPrPartition type also holds the actual messages (as defined in 8.1.7).

FetchRequest

Request Payload for Fetch API. Fetchs chunk of one or more logs for some topic-partitions. Defines a starting offset at which to begin fetch.

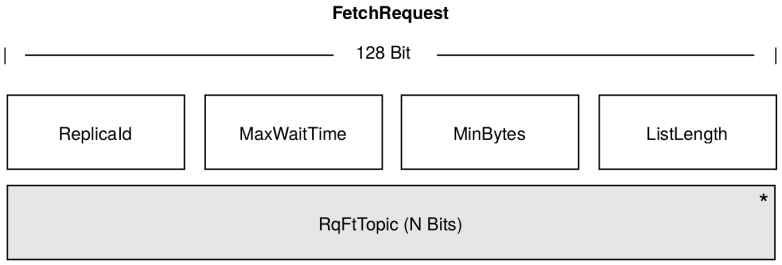


Figure 8.5.: Format of type FetchRequest

ReplicaId	Data.Word32	Indicates the node id of the replica initiating this request. (Not yet used in broker implementation)
MaxWaitTime	Data.Word32	Maximum amount of time in milliseconds to block in order to wait if insufficient data is available at the time the request is issued.
MinBytes	Data.Word32	Minimum number of bytes of messages that must be available to provide a response.
ListLength	Data.Word32	Number of elements in the list of topics.
[RqFtTopics]	Data.List	Sequence of FtTopics (see below).

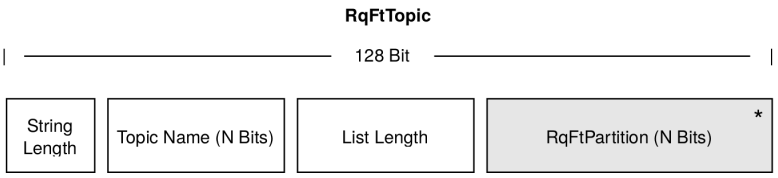


Figure 8.6.: Format of type RqFtTopic

StringLength	Data.Word16	Length in bytes of the string TopicName.
TopicName	Data.ByteString	Name of the topic to be fetched.
ListLength	Data.Word32	Number of elements in the following list.
[RqFtPartition]	Data.List	Sequence of FtPartitions, see below.

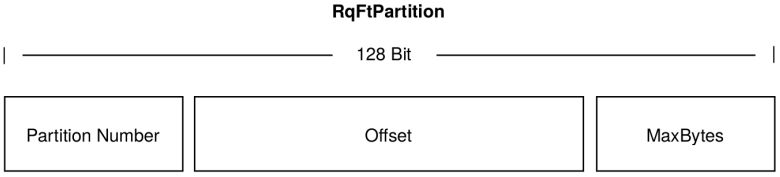


Figure 8.7.: Format of type RqFtPartition

PartitionNumber	Data.Word32	Id of the partition to be fetched.
Offset	Data.Word64	Offset to begin the fetch from.
MaxBytes	Data.Word32	Minimum number of bytes of messages that must be available to provide a response.

Metadata Request

The request format for the metadata is fairly simple. The RequestMessage header, containing the appropriate API key, leads to initiate the broker to reponse meta information about existing topics, partitions and brokers. The request can contain an optional list of topics, for which explicit information is fetched.

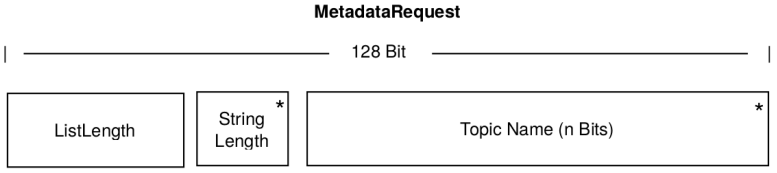


Figure 8.8.: Format of type MetadataRequest

ListLength	Data.Word32	Number of elements in the list of StringLength and TopicName.
StringLength	Data.Word16	Length in bytes of the string TopicName.
TopicName	Data.ByteString	The topic to produce metadata for. If empty the request will yield metadata for all topics

8.1.6. Types related to Response

A response message is the on-wire format of data which is being sent from broker to client.

Each request message has the same header fields for which API they are used. The real payload of the request is individual to the particular API which is used. A response will always match the paired request (e.g. a ProduceResponse is sent in return to a ProduceRequest).

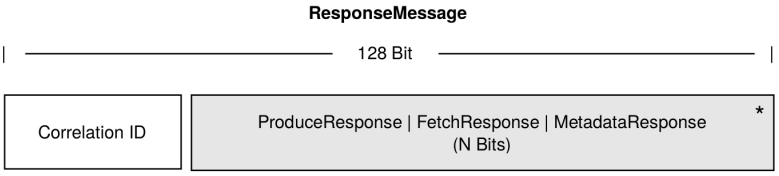


Figure 8.9.: Format of type ResponseMessage

CorrelationId	Data.Word32	Id that is supplied by request is passed back (for request-reply reference).
[Response]	Data.List	Sequence of Responses of the same type (either ProduceResponse, FetchResponse or others).

ProduceResponse

Response Payload for Produce API. Sends an ErrorCode which determines whether a produce request for a specific topic-partition succeeded (ErrorCode 0) or ended in a particular failure, specified by the error code.

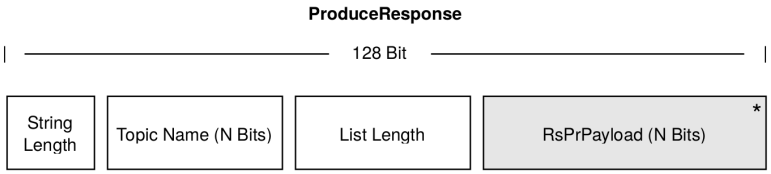


Figure 8.10.: Format of type ProduceResponse

StringLength	Data.Word32	Length in bytes of the string TopicName.
TopicName	Data.ByteString	Name of the topic the response entry corresponds to.
ListLength	Data.Word32	Number of element in the list of payload sequences.
[RsPrPayload]	Data.List	Sequence of RsPrPayload, see below.

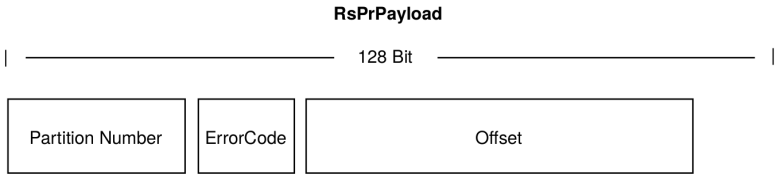


Figure 8.11.: Format of type RsPrPayload

PartitionNumber	Data.Word32	Number of the partition this response entry corresponds to.
ErrorCode	Data.Word16	Topic-Partition-specific error code. Error code 0 determines that data has been published successfully.
Offset	Data.Word64	See protocol definition.

FetchResponse

Response Payload for Fetch API. Sends requested chunk of one or more MessageSet's (see 8.1.7).

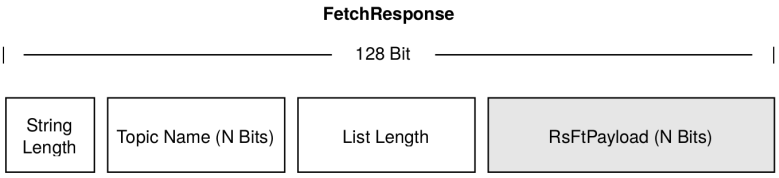


Figure 8.12.: Format of type FetchResponse

StringLength	Data.Word32	Length in bytes of the string TopicName.
TopicName	Data.ByteString	Name of the topic the response entry corresponds to.
ListLength	Data.Word32	Number of elements in the list of payload entires.
[RsFtPayload]	Data.List	Sequence of RsPrPayload, see below.

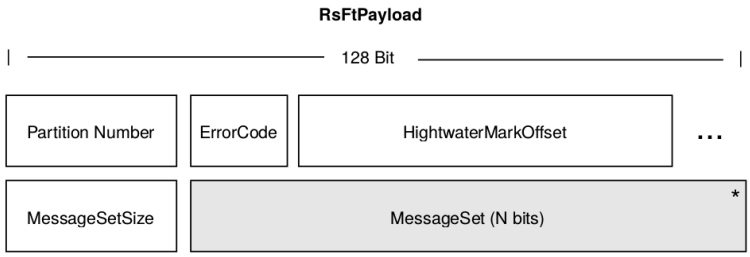


Figure 8.13.: Format of type RsFtPayload

PartitionNumber	Data.Word32	Number of the partition the response entry corresponds to.
ErrorCode	Data.Word16	Topic-Partition-specific error code. (Error code 0 determines that data has been published successfully).
HighwaterMarkOffset	Data.Word64	The offset at the end of the log for this partition. This can be used to determine how many messages the client is behind the end of the log.
MessageSetSize	Data.Word32	The size in bytes of the message set for this topic and partition.
[MessageSet]	Data.List	The message data fetched from this partition (format described below).

Metadata Response

As simple the metadata request is, the more complex the response format turns out. It contains information about provided brokers, topics, partitions and their replication status.

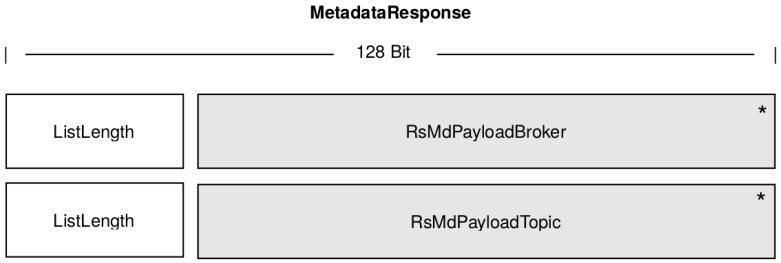


Figure 8.14.: Format of type MetadataResponse

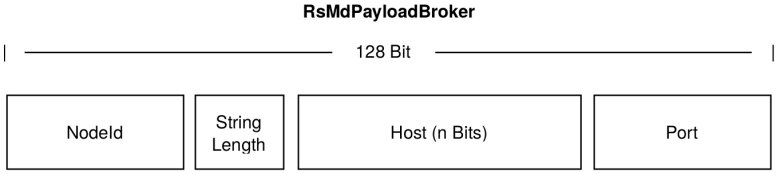


Figure 8.15.: Format of type RsMdPayloadBroker

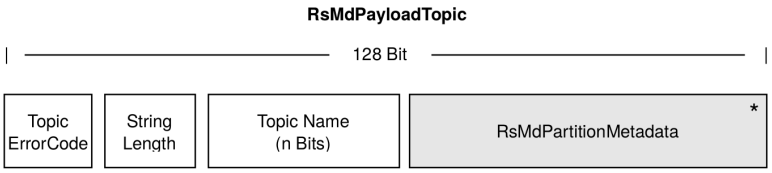


Figure 8.16.: Format of type RsMdPayloadTopic

ListLength	Data.Word32	Number of elements in the following list of broker informations.
[RsMdPayloadBroker]	Data.List	Sequence of RsMdPayloadBroker, contains node Id, hostname and port information of broker.
ListLength	Data.Word32	Number of element in following list of topic informations.
[RsMdPayloadTopic]	Data.List	Sequence of RsMdPayloadTopic, contains all, or the requested topic names with approriate error code which determines status. Also contains a sequence of RsMdPayloadMetadata (see below).

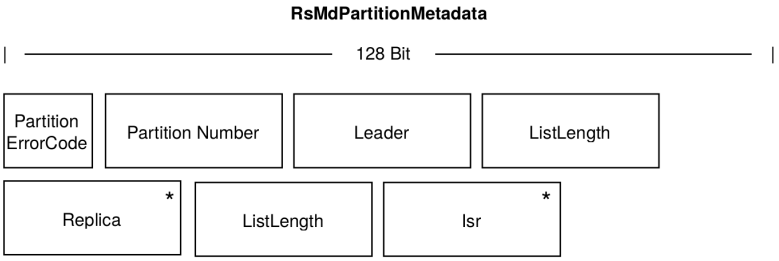


Figure 8.17.: Format of type RsMdPartitionMetadata

PartitionErrorCode	Data.Word16	Partition specific error code.
PartitionNumber	Data.Word32	Number of Partition the metadata is for.
Leader	Data.Word32	The node id for the broker currently acting as leader for this partition. If no leader exists (because when beeinh in the middle of a leader election) this id will be -1.
ListLength	Data.Word32	Number of element in following list of replicas.
[Replicas]	Data.List	The set of alive nodes that currently acts as slaves for the leader for this partition, given as Data.Word32.
ListLength	Data.Word32	Number of element in following list of in-sync-replicas.
[Isr]	Data.List	The set of the replicas that are "caught up" to the leader, given as Data.Word32.

8.1.7. Types related to Data

The transported data holds a common structure, called a MessageSet. This format happens to be used both for the on-disk storage on the broker as well as the on-the-wire format. Therefore the broker do not need to perform any transformations to the actual message when persisting to the log or reading from it.

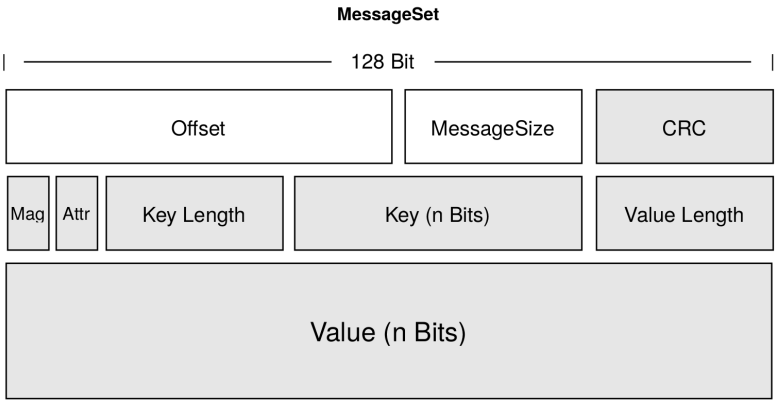


Figure 8.18.: Format of type MessageSet including type Message and Payload

Offset	Data.Word64	Determines log offset number in log. When the producer is sending messages it doesn't actually know the offset and can fill in any value here it likes.
MessageSize	Data.Word32	Determines size of Message in Bytes.
Message		Message format, see below.

Message

Message is part of the MessageSet and contains a checksum to check integrity of the actual payload of the message. To abstract the fields which are checked we combine them in another type (see Payload).

CRC	Data.Word32	The CRC is the CRC32 checksum. This is used, to check the integrity of the message on the broker and consumer.
Payload		Payload format (see below).

Payload

This types defines the actual payload of a message along with its meta information.

Magic	Data.Word8	This is a version id used to allow backwards compatible evolution of the message binary format. The current value is 0.
Attributes	Data.Word8	This byte holds metadata attributes about the message. The lowest 2 bits, contain the compression codec used for the message. The other bits, should be set to 0 (not defined yet).
KeyLen	Data.Word32	Determines length of Key field in bytes.
Key	Data.ByteString	The key is an optional message key that is used for partition assignment. The key can be null.
ValueLength	Data.Word32	Determines length of Value field in bytes.
Value	Data.ByteString	The actual message contents as a byte array. The message can be null. Kafka also supports recursive messages in which case the value may itself contain a message set. This functionality is not implemented yet.

8.2. Encode / Decode

There are libraries in Haskell to encode or decode data to or from binary formats. As for binary serialization, the Data.Binary[6] library is being used as it works with lazy bytestrings. As an alternative to Data.Binary, the *cereal* library (Data.Serialize[7]) also provides binary serialization capabilities – especially for strict bytestrings. Lazy bytestring represent a list of bytes, this plays nicely along with streaming and thus for the use case of a message broker. Additionally, lazy bytestrings support appending in O(1) complexity which will be used frequently in the serialization process. This makes further usage of the ByteString slightly more efficient.

Parsing libraries such as Attoparsec[8] or Parsec[9] where considered but proved to not be suitable for performance-critical applications. In fact, there is no need for parsing functionality – such as syntactic analysis – regarding the tasks the broker has to be capable of.

8.2.1. Decode Request/Response

As mentioned in previous sections, every request or response has the same header fields. Functions are provided to either parse a request or response and remind of the original definition of the Kafka protocol as grammar. To identify the actual type of request, the protocol describes an API Key field which holds a numeric code which explicitly defines the type of request. Depending on the API key, the appropriate parse function will be called.

[6] https://hackage.haskell.org/package/binary-0.4.1/docs/Data-Binary.html
[7] https://hackage.haskell.org/package/cereal-0.4.1.1/docs/Data-Serialize.html
[8] https://hackage.haskell.org/package/attoparsec
[9] https://hackage.haskell.org/package/parsec

The Get Monad (Data.Binary.Get) allows to comfortably parse ByteString in its big endian network order and allows to decode to appropriate types.

The following code shows the function for parsing an arbitrary request. The parse function for the appropriate request is called by pattern matching the api key.

```
import Data.Binary.Get

requestMessageParser :: Get RequestMessage
requestMessageParser = do
    apiKey        ← getWord16be
    apiVersion    ← getWord16be
    correlationId ← getWord32be
    clientIdLen   ← getWord16be
    clientId      ← getByteString $ fromIntegral clientIdLen
    request       ← case (fromIntegral apiKey) of
        0 → produceRequestParser
        1 → fetchRequestParser
        3 → metadataRequestParser
        -- further API Codes
        _ → -- Invalid API Key
    return $ RequestMessage apiKey apiVersion correlationId clientIdLen
    clientId request
```

Listing 8.3: Decode of RequestMessage, based on ApiKey

Batched sequences are decoded as Data.List where the following function provides a generic way to decode a list with a given size and parse each element:

```
parseList :: Int → (Get a) → Get [a]
parseList i p = do
  if (i < 1)
    then return []
    else do x ← p
            xs ← parseList (i-1) p
            return (x:xs)
```

Listing 8.4: Decode list of any type

A batched sequence of MessageSets is a special case where the protocol does not provide a field which determines the number of elements. There is just a field which contains the size in bytes of all message sets. Therefore we implemented a list decoder for sequences of MessageSets:

```
parseMessageSets :: Int → Get [MessageSet]
parseMessageSets i = do
    if (i < 1)
    then return []
    else do x ← messageSetParser
            xs ← parseMessageSets $ i - (fromIntegral $ length $ buildMessageSet x)
            return $ (x:xs)
```

Listing 8.5: Decode list of MessageSet's

Decoding a ByteString can be done with `runGet f b`, where `f` declares the parser function (indicated through Get a return type), and `b` the bytestring to decode. This construct implies that the Get monad is applied at the latest possible point in time in constant complexity. For example:

```
1 import Data.Binary.Get
2
3 decodeRsMessage :: B.ByteString → ResponseMessage
4 decodeRqMessage b = runGet requestMessageParser b
```

8.2.2. Encode Request/Response

Obviously the encode module provides the opposite functionalities as the decode module and was separated just for clarity. The implementation also relies on Data.Binary[10] and uses the Put Monad (Data.Binary.Put) to construct ByteStrings.

The following code shows the function for encoding an arbitrary request:

```
1  buildRqMessage :: RequestMessage → Put
2  buildRqMessage e = do
3    putWord32be $ rqSize e
4    putWord16be $ rqApiKey e
5    putWord16be $ rqApiVersion e
6    putWord32be $ rqCorrelationId e
7    putWord16be $ rqClientIdLen e
8    putByteString $ rqClientId e
9    case (fromIntegral $ rqApiKey e) of
10     0 → buildProduceRequest  $ rqRequest e
11     1 → buildFetchRequest    $ rqRequest e
12     3 → buildMetadataRequest $ rqRequest e
13     -- further API Codes
14     _ → -- Invalid API Key
```

Listing 8.6: Encode RequestMessage, depending on ApiKey

Batched sequences are encoded as Data.List whereas the following function provides a generic way to build the list in a given size and encode each element:

```
1 buildList :: (a → Put) → [a] → Put
2 buildList builder [] = putLazyByteString BL.empty
3 buildList builder [x] = builder x
4 buildList builder (x:xs) = do
5   builder x
6   buildList builder xs
```

Listing 8.7: Encode list of any type

Encoding a ByteString can be done with `runPut f t`, whereas `f` declares the build function (indicated through Put return type), and `t` the type to decode. Like with decoding, this construct implies that the Put monad is applied at the latest possible point in time in constant complexity. For example:

[10] https://hackage.haskell.org/package/binary-0.4.1/docs/Data-Binary.html

```haskell
import Data.Binary.Put

enodePrRqMessage :: RequestMessage → B.ByteString
enodePrRqMessage r = runPut buildPrRqMessage r
```

8.3. Client Library

As part of the protocol implementation, the client library provides an API for clients to interact with the protocol. The goal is to isolate the producer and consumer clients from implementation details such the types and encode/decode functions.

8.3.1. Types

The types provided by the client library give a simplified abstraction for the types of the protocol implementation. All values which need to be provided by a client are covered. Every type which is irrelevant to the client is hidden.

Depending on the API a client wants to access, either a `Produce`, `Fetch` of `Metadata` type needs to be defined. Each of them consists of a `Head`, which contains the header information every request needs to include. Furthermore, the library types allow the client to define nested sequences of topics and partitions to fully support the batching approach of the Kafka protocol.

```haskell
data Req = Produce Head [ToTopic] | Fetch Head [FromTopic] | Metadata Head [OfTopic]
data Head = Head
  { apiV      :: Int
  , corr      :: Int
  , client    :: BS.ByteString
  }
```

Listing 8.8: Simplified types for client API

Example construct of nested produce request with multiple topics and partitions:

```
Produce
    Head 0 0 clientId
    [ToTopic topicA [ToPart 0 [ms], ToPart 1 [ms]],
     ToTopic topicB [ToPart 0 [ms]]]
```

8.3.2. Send Request

```haskell
class Packable a where
  pack :: a → RequestMessage

instance Packable Req where
  pack (Produce head ts)  = packPrRqMessage head ts
  pack (Fetch head ts)    = packFtRqMessage head ts
  pack (Metadata head ts) = packMdRqMessage head ts
```

Before sending an API call to the broker, the defined types above need to be packed (see listing above) into the actual types of the Kafka protocol implementation (with all the additional information which are hidden from the client). Therefore the client library exposes a `sendRequest` function. It packs the provided `Req` to the protocol-conform `RequestMessage` and encodes it to ByteString. Finally, the binary data is sent to the given socket.

```
1 sendRequest :: Socket → RequestMessage → IO ()
2 sendRequest socket requestMessage = do
3     SBL.sendAll socket msg
4     where msg = runPut $ buildRqMessage $ pack requestMessage
```

Listing 8.9: Pack and send function of client API

8.3.3. Decode Response

For parsing received responses sent by the broker, the client library exposes functions for decoding the binary data. To expose functionality for the client to decode responses sent from broker, the client library provides specific decode functions:

```
1 decodePrResponse :: BL.ByteString → ResponseMessage
2 decodePrResponse a = runGet produceResponseMessageParser a
3
4 decodeFtResponse :: BL.ByteString → ResponseMessage
5 decodeFtResponse b = runGet fetchResponseMessageParser b
6
7 decodeMdResponse :: BL.ByteString → ResponseMessage
8 decodeMdResponse b = runGet metadataResponseMessageParser b
```

Listing 8.10: Exposed decode functions of client API

Instead of exposing the decode functions for response, it is conceivable to provide unpack functionality which decodes the request and extracts only relevant fields to a simplified type structure (analog to sendRequest function above). This is not implemented yet.

8.3.4. Client Examples

This example shows a very rudimentary console client which either can act as producer by sending a producer request or as a consumer by sending a fetch request. After sending a request the client waits for a corresponding response from the broker. It need to be considered that the following code still misses a lot of functionality to act as reasonable client. It only demonstrates and proves the use of the client library quite simplified.

First, the client initializes a stream socket an open connection to a given host address and port (broker is running). After the broker accepted the connection the client is ready so send requests:

```
import qualified Kafka.Client as C
import qualified Data.IP as IP
import Network.Socket

main = do
 sock ← socket AF_INET Stream defaultProtocol
 setSocketOption sock ReuseAddr 1
 let ip = IP.toHostAddress (read "127.0.0.1" :: IP.IPv4)
 connect sock (SockAddrInet 4343 ip)
```

Listing 8.11: Initialize client by connceting to broker

Second step is to request metadata from the broker. Therefore the client builds a metadata request, send it to the broker an receives the corresponding response:

```
clientId ← getLine
let requestHeader = C.Head 0 0 (C.stringToClientId client)

let mdReq = Metadata requestHeader [] -- get metadata for all provided topics
sendRequest sock $ (pack mdReq)
mdInput ← SBL.recv sock 4096
let mdRes = decodeMdResponse mdInput
```

Listing 8.12: Retrieve metadata from broker

Producer Client

The following example of a producer client first retrieves topic name and partition the messages are publibshed to from console. After, each given console input is send to the broker as single request. After sending a request the clients receives and decodes the response.

```
t ← getLine
partition ← getLine
let p = read partition :: Int

 forever $ do
   putStrLn "Insert message"
   input ← getLine
   let prReq = C.Produce
                  requestHeader
                  [ C.ToTopic (C.stringToTopic t) [ C.ToPart p
                  [(C.stringToData input)]]]
   C.sendRequest sock $ prReq

   input ← SBL.recv sock 4096
   let res = C.decodePrResponse input
   print res
```

Listing 8.13: Producer client example

Consumer Client

Send a fetch request with given offset, topic and partition constantly every one second to the broker. Paralell it receives and decodes incoming responses.

```
putStrLn "Give Topic Name"
  topicName ← getLine
  partition ← getLine
  let p = read partition :: Int
  fetchOffset ← getLine
  let o = read fetchOffset :: Int

  let ftReq = Fetch requestHeader [ FromTopic C.stringToTopic t [ FromPart p o ]]
  forever $ do
    sendRequest sock ftReq
    forkIO $ do
      response ← SBL.recv sock 4096
      let res = decodeFtResponse response
    threadDelay 1000000
```

Listing 8.14: Consumer client example

8.4. Testing

The Haskell community offers multiple frameworks for automated testing. To test the functionality of the protocol implementation, hspec[11] is used. It provides sustainable integration possibilities and generates useful tests logs to `/dist/test/`. It also supports interoperability for other frameworks like HUnit[12].

The main purpose within the implemented tests is to determine if the encode and decode functions are corresponding. Therefore a given type (e.g. ProduceRequest) is encoded to bytestring and afterwards decoded back to the type. If the resulting type is equals to the original, it means the process was successful. This helps to expose wrong or missing instruction in encoding/decoding when protocol changed. Another goal is to test abnormal input or, for example, varying of the size of lists from zero to N elements. To simplify the code of single tests, type fixtures are implemented. The fixtures helps to assemble types together from different fixtures when their fields are already filled with some sample values.

As for now, not the whole protocol implementation is covered by test. The implemented tests still give a boilerplate for further implementation. The following code shows an extract of the existing protocol tests:

```
1  import Test.Hspec
2  import SpecFixtures
3
4  spec :: Spec
5  spec = do
6    describe "Kafka.Protocol" $ do
7      context "MessageSet" $ do
8        it "Test1: Default Value" $ do
9          let ms = getMessageSetFixture $
10                    getMessageFixture $
11                    getPayloadFixture "Test"
12         testMs ms
13       it "Test2: No Value" $ do
14         let ms = getMessageSetFixture $
15                    getMessageFixture $
16                    getPayloadFixture ""
17         testMs ms
18       it "Test3: JSON like Value" $ do
19         let ms = getMessageSetFixture $
20                    getMessageFixture getPayloadFixture
21            "[ obj : { value : 123 }]"
22         testMs ms
23
24 testMs :: MessageSet → Expectation
25 testMs m = (runGet messageSetParser $ runPut $ buildMessageSet m)
26               `shouldBe` m
```

Listing 8.15: Extract from protocol tests with hspec

[11] https://hackage.haskell.org/package/hspec
[12] https://hackage.haskell.org/package/HUnit

9. Implementation Broker

The broker implementation is a stand-alone server application based on the Apache Kafka protocol implementation (see chapter 8). To clarify the program flow from receiving data to the network, to handling the requests, and to accessing the file system, the broker is split into three layers, each with its own module.

Network Layer
: This module encapsulates action on the network. It initiates connections and receives bytes from client. After the received bytes are chunked into single requests and provide it to the API Layer.

API Layer
: This module handles incoming requests. It first parses the received bytes and then proceeds to an appropriate action, depending on the delivered API key. After performing the action, an appropriate response is generated and provided to the Network Layer to send back to the client.

Log Layer
: This module encapsulates actions to the log on the filesystem. Fundamental functions are appending messages to the log or reading from it.

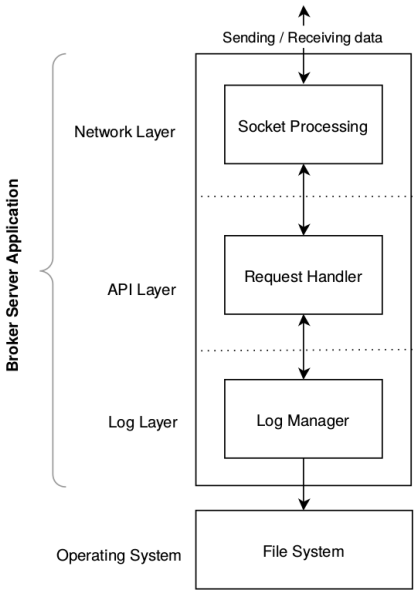


Figure 9.1.: Three layer of broker server application

9.1. Server Architecture

The server architecture concept of the broker application includes the following components working as own thread instances:

Main Thread (single instance)
: The main function of the broker bootstraps the whole server application. It first initializes the network layer and then forks the *Acceptor*, *Responder* and *API Handler* threads. Started threads are managed with the `withAsync` function of the Control.Concurrent.Async[1] library. It adds a thin layer over the basic concurrency operations provided by Control.Concurrent[2] and gives the ability to wait for the return value of a thread. This provides additional safety and control if a thread crashes.

Acceptor Thread (single instance), Network Layer
: This thread accepts new client connections (see 9.2.1) in a running loop. To support multiple connections from different clients, it forks a new thread for processing incoming data whenever a new connection is established.

Connection Processor Thread (instance per connection), Network Layer
: The incoming data needs to be processed as fast as possible. If the limit of the socket buffer is reached, the throughput on the network drops dramatically. The processor thread constantly receives requests from a given connection (see 9.2.2) and provides it to the API handler thread. If the connection is closed, the thread will return.

Responder Thread (single instance), Network Layer
: This thread works off the provided responses and sends them back to the original client (see 9.2.3). As for now, there is only one instance of this thread. Further one instance per connection should be forked. The challenge then would be to stop a responder threads properly after a connection to a client is closed.

API Handler Thread (one instances)
: The API handler is a worker thread which works off the received request. It also builds appropriate responses and provides it to the responder thread (see 9.3.1). It is conceivable to run more than a single instance of the API handler. Due to advanced synchronisation, this is not realized yet.

Original Apache Kafka has a fix and configurable amount of threads handling network requests by working with thread pools. Thanks to the optimizations of the GHC IO manager, the communication with the Haskell threads and the OS is very efficient due to thread multiplexing. Thread pooling has no further advantages than one socket processor thread instance per connection.

For transfer of data between the layers, channels are implemented. The Control.Concurrent.Chan[3] package provides a one-way FIFO communication channel. This concept is used to separate the three layers from each other. The fact that *chan* is unbounded brings a risk. While *writeChan*–which is being used to write to a channel–succeeds immediately, there is a chance that the consuming thread is not able to read the same amount of data in a given time and thus the channel will grow in an unchecked manner. [30]

[1] https://hackage.haskell.org/package/async-2.0.0.0/docs/Control-Concurrent-Async.html
[2] https://hackage.haskell.org/package/base-4.5.0.0/docs/Control-Concurrent.html
[3] http://hackage.haskell.org/package/base-4.8.0.0/docs/Control-Concurrent-Chan.html

To isolate the network layer from any build or parse process, the transfered data is of type `Data.ByteString`. Because the response to a particular request needs to be sent back over the same connection, the data passed to a channel is defined as tuple of the received bytes and the corresponding socket connection information:

```
1 import qualified Data.ByteString.Lazy as B
2
3 type ChanMessage = ((Socket, SockAddr), B.ByteString)
4 type RequestChan = Chan ChanMessage
5 type ResponseChan = Chan ChanMessage
```

Listing 9.1: Initialize channels for threading

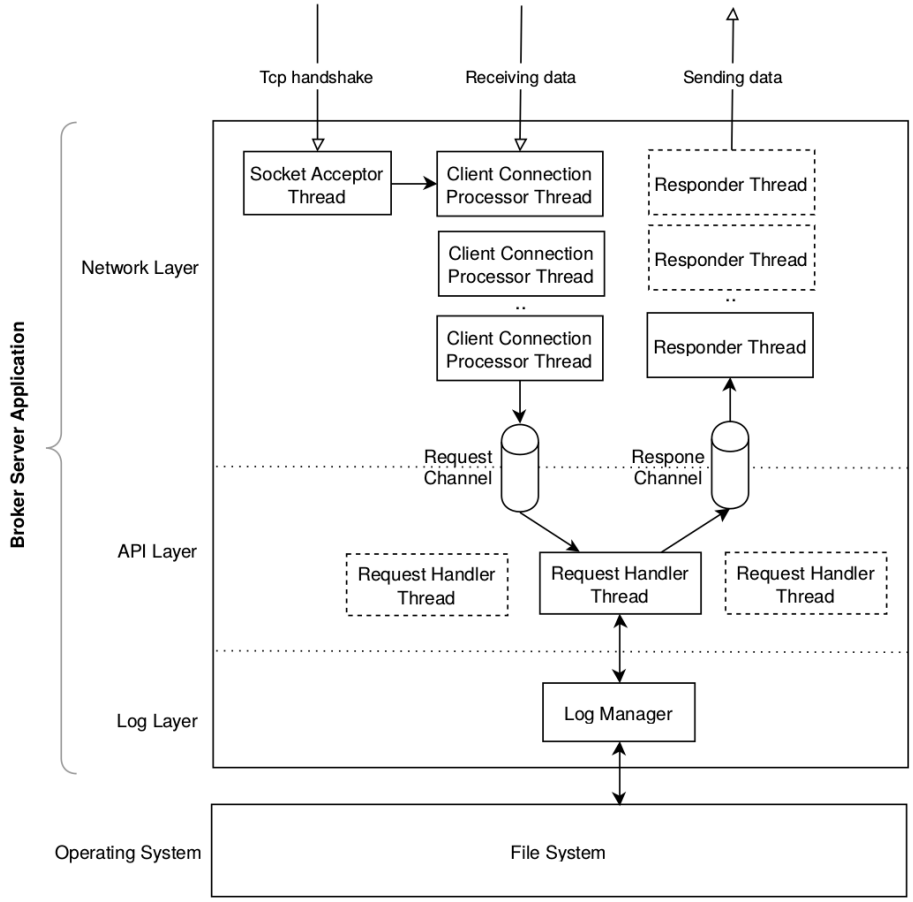


Figure 9.2.: Internals of broker server application

9.2. Network Layer

The broker application has a thin network layer which is responsible for receiving and sending binary data to clients. Instead of using HTTP, connection-oriented sockets (TCP) are used. Therefore, the broker is independent from any HTTP implementation, and clients can make use of advanced TCP features like the ability to simultaneously poll many connections or multiplex requests. While a socket is an endpoint of a bidirectional inter-process communication flow, each connection established to the broker is basically a socket connection. That being said, it is important to provide a reliable socket server implementation which will serve correctly under a heavy load. Haskell provides full control over sockets using the Network.Socket[4] library which exposes the C socket API[5].

This section shows the basic operations and describes the used libraries in the network layer. For further details about the threading concept, see section 9.1.

9.2.1. Socket Connection Establishment

Initialize Socket

First of all, the initialization process of a socket happens on the server by providing the following configuration parameters:

- **Protocol Family:** *AF_INET*, for network protocol IPv4.
- **Socket Type:** *Stream*, which provides sequenced, reliable, two-way, connection-based byte streams.
- **Protocol Number:** *0*, which indicates that the caller does not want to specify the protocol, leaving it up to the service provider.

Bind Socket

After the configuration is set, the socket has to be associated with an address structure which is a constellation of an IP Address and a Port. The constructor *SockAddrInet* of data type *SockAddr* takes the following two arguments:

- **Port Number** 4343 (As for now static, but could be done with configuration)
- **Host Address** iNADDR_ANY, which binds the socket to all interfaces.

Listen

While the socket is created and bound to an interface, the socket state can be entered into a listening state. The only configuration parameter which has to be considered is the maximum number of queued connections requesting to be accepted (also called backlog). While this parameter is not critical in the constellation of this broker, we set the queue length to **50** which is also the default value in the Java SocketServer implementation. *Note that the focus remains on the amount of data being processed rather than the number of clients being served.*

[4] https://hackage.haskell.org/package/network-2.3.0.7/docs/Network-Socket.html
[5] http://pubs.opengroup.org/onlinepubs/7908799/xns/syssocket.h.html

9.2.2. Receive Data

To receive data from the socket buffer, the function `recv` from Network.Socket.ByteString.Lazy[6] library which provides access to the Unix socket interface is used. Because the underlying protocol is in binary format, and the incoming data later need to be parsed directly from type `ByteString` this module is more efficient than the string based network functions. We took the lazy variant of the library because the input gets directly parsed by our lazy based encoder.

Because each incoming request needs to be handled individually, the exact amount of bytes to get one particular request from the socket buffer first need to be determined. Therefore the first four bytes—which defined the request size according to—the protocol, need to be received and parsed to an numeral value.

```
1  import qualified Network.Socket.ByteString.Lazy as S
2  import qualified Data.ByteString.Lazy as BL
3
4  recvFromSock :: (Socket, SockAddr) → IO BL.ByteString
5  recvFromSock (sock, sockaddr) =  do
6      respLen ← S.recv sock (4 :: Int64
7      let parsedLen = getLength respLen
8      req ← recvExactly sock parsedLength
9      where
10         getLength = runGet $ fromIntegral <$> getWord32be
```

Listing 9.2: Receiving a request from socket

After getting the size of a particular request, the evaluated amount of bytes is received from the socket. Because of the blocking semantics of Unix sockets and TCP packet fragmentation, it is not guaranteed that the `len` argument for the `recv` system call gets exactly the number of bytes requested. The call may produce less data than specified. Therefore the following function is implemented to get the data in chunks until the entire request is read.

```
1  recvExactly :: Socket → Int64 → IO BL.ByteString
2  recvExactly sock size = BL.concat ∘ reverse <$> loop [] 0
3    where
4      loop chunks bytesRead
5          | bytesRead ≥ size = return chunks
6          | otherwise = do
7              chunk ← S.recv sock (size - bytesRead)
8              if BL.null chunk
9                then return chunks
10               else loop (chunk:chunks) $! bytesRead + BL.length chunk
```

Listing 9.3: Receive exactly amount of bytes from socket

[6] https://hackage.haskell.org/package/network-2.3.0.1/docs/Network-Socket-ByteString-Lazy.html

9.2.3. Send Data

For every processed request, an appropriate response is sent back to the client. The network layer does not need to generate responses, actually they are provided as binary data from the API layer. The network layer simply needs to send the bytes over the existing connection back to the client. For sending data over the socket, function `sendAll` from Network.Socket.ByteString.Lazy[7] is used. This function continues to send data until either all data has been sent or an error occurs. For further details about Error Handling, see section 9.4.

```
1 import qualified Data.ByteString.Lazy as B
2 import qualified Network.Socket.ByteString.Lazy as S
3
4 sendResponse :: (Socket, SockAddr) → B.ByteString → IO (Either IOError ())
5 sendResponse (socket, addr) responsemessage =
6               try(S.sendAll socket responsemessage) :: IO (Either IOError ())
```

Listing 9.4: Send response back to client via socket

9.3. API Layer

This part of the broker handles incoming requests, initiates any actions needed, and prepares appropriate responses. It fulfills the essential part in error handling which is described in section 9.4.

9.3.1. Decode Request

After getting the next bytestring from network layer, the incoming request first gets decoded. For this part, the API layer simply uses functions provided by the protocol implementation (see chapter 8).

9.3.2. Handling Requests

The main focus of the API layer lies on handling the requests properly. Depending on the API key field of type `RequestMessage`, the request handler determines an appropriate action. In case of a valid API key, an action is performed and a encoded response message is passed back to the Network Layer.

For nearly all provided APIs, the broker needs access to the functions provided by the log subsystem as described in section 9.5. Therefore, the API layer initializes the LogManager and passes it to the handling function below:

[7] https://hackage.haskell.org/package/network-2.3.0.1/docs/Network-Socket-ByteString-Lazy.html

```haskell
handleRequest :: RequestMessage → LogManager.State
                 → IO (Either HandleError BL.ByteString)
handleRequest rm = do
   handle ← case rqApiKey rm of
    0  → handleProduceRequest rm
    1  → handleFetchRequest rm
    2  → handleOffsetRequest rm
    3  → handleMetadataRequest rm
    8  → handleOffsetCommitRequest rm
    9  → handleOffsetFetchRequest rm
    10 → handleConsumerMetadataRequest rm
    _  → return (Left UnknownRqError)
   return handle
```

Listing 9.5: Handling requests depending on ApiKey

HandleProduceRequest

Handling a produce request implies that the API layer passes the containing messages to the append function provided by the `LogManager`. A request may contain messages for multiple topics or partitions, and each combination refers to another log file. Therefore, the job of the API layer is to unnest the request into single calls to the `LogManager`.

```
Simplified structure of nested sequences in a produce request:
[Topic [Partition [MessageSet]]]

Example:
[TopicA [Part0 [M], Part1 [M]], Topic B [Part0 [M]]]

Unnested to three tuples which can be feed to LogManager as a single call:
[(TopicA, Part0, [M]), (TopicA, Part1, [M]), (TopicB, Part0, [M])]
```

Thanks to Haskell's *list comprehension*, the unnesting of the request can be made quite elegant:

```haskell
[ (rqToName x, rqPrPartitionNumber y, rqPrMessageSet y ) | x ← rqPrTopics req, y ← rqToPartitions x ]
```

Listing 9.6: List comprehension for unnesting produce request

HandleFetchRequest

Analog to the produce request, handling a fetch request leads to an according `readLog` call on the `LogManager`. The retrieved messages are packed and encoded to a protocol conform fetch response.

Fetching data for multiple topics or partitions is not yet supported.

HandleMetadataRequest

Handling a request to the metadata API includes gathering information on the broker system and provided topics and packing them to a metadata response. Getting replication and partition related information is not yet implemented.

For retrieving a list of provided topics, the `LogManager` is again involved. The exposed function `getTopicNames` analyses the existing file structure and gives the available topics.

Information regarding the broker host includes local IP. Therefore, the library Network.Info[8] is used:

```
1 import qualified Network.Info as N
2 getHost :: IO N.NetworkInterface
3 getHost = do
4   ns ← N.getNetworkInterfaces
5   return $ head $
6         filter ((N.NetworkInterface name ipv4 ipv6 mac) → name ==
  "eth0") ns
```

Listing 9.7: Example of getting the brokers host adress

[8] https://hackage.haskell.org/package/network-info

9.4. Error Handling

A message broker relies fundamentally on error handling, as fault tolerance and reliability are key requirements of such a system. The first part of this section concentrates on the architectural details and decisions made regarding error handling. Using a demonstration of a message flow, possible edge cases are considered and uncovered in order to provide reliability to the user of this broker. The second part covers the techniques used in Haskell to handle the previously described concept and cases where errors may occur. It will also involve part of the concept behind the error handling of Apache Kafka, which is partly integrated in the Apache Kafka Protocol and thus adapted in this implementation.

9.4.1. Message Flow

The following figure (9.3) provides insight into the process that an incoming request message goes through. For simplicity, one may think of a message proceeding through all steps successively. But in fact, multiple requests are handled at the same time. Assuming the connection to the broker is established and the client is ready to send requests to the broker, there are principally two types of errors—SocketError and RequestError—that may occur for each, which there is a separate handler to handle errors appropriately.

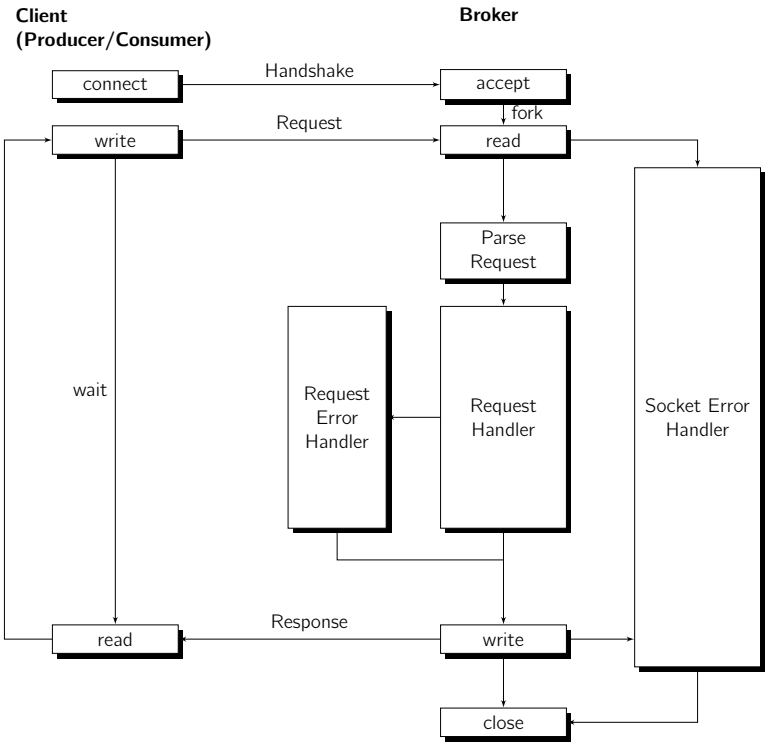


Figure 9.3.: Broker error handling concept

Socket Error

During the process of listening to an open connection and reading an incoming byte-stream, as described in 9.2, there is a chance of unexpected errors occuring. In this case, the underlying C socket implementation will return a result of *-1* which will then result in an IOError[9].

Errors at this stage will be handled using the *Socket Error Handler* and do not result in any response to the client who initiated the request that resulted in an error, as the socket connection may be broken. Instead, the error is simply caught and reported as console output. Alternatives would be using a separate logging framework such as hslogger[10]. At worst, the socket connection is closed.

Request Error

One step further in the API-Stage (9.3), past where socket errors may occur, the heavy part of error handling begins. Each step the request passes may result in an error. To name a few, this can be the case starting by parsing the request, writing data to the disk executed by the log subsystem, or at last producing the response message. All of the mentioned cases and others who fall into the related category will be handled by the *RequestErrorHandler*, where a set of possible errors or category of error is defined, resulting in an appropriate response message sent to the client.

9.4.2. Defining Error Types

The previously introduced concept of the two error handlers, each responsible for a certain layer of the application, can be built in Haskell conveniently by using the Either Monad[11] and *pattern matching* of custom defined error types. This will allow taking actions based on the given error type.

Error types related to the networking layer are simply distinguished between an error occurring either in the receiving or responding process.

```
1 data SocketError =
2       SocketRecvError String
3       | SocketSendError String
4       deriving Show
```

Listing 9.8: SocketError types

Any further details are not separated by sub-types. Instead, they can be described in the first argument of the `SocketError`, namely a **String**.

To handle errors during the process of request handling, the data type `HandleError` contains specific types for each edge case in any part of the application. Those types can be considered as an interface for errors between the request handler and the underlying subsystems. This types are only defined for the API layer where the handling of requests occurs.

[9] https://hackage.haskell.org/package/base-4.4.1.0/docs/System-IO-Error.html
[10] https://hackage.haskell.org/package/hslogger
[11] https://hackage.haskell.org/package/category-extras-0.52.0/docs/Control-Monad-Either.html

```
data HandleError =
        PrWriteLogError Int String
      | PrPackError String
      | ParseRequestError String
      | FtReadLogError Int String
      | SendResponseError String
      | UnknownRqError
        deriving Show
```

Listing 9.9: Handle Error types

9.4.3. Error Handlers

With the knowledge about the types they are related to, either socket or request errors, it is now possible to create handler functions. They act as a central place where any kind of error can be sent and will provide the functionality to appropriately handle. Figure 9.4 illustrates that the two mentioned error handlers distinguish their next action based on the data type containing its arguments that is sent to the error handler.

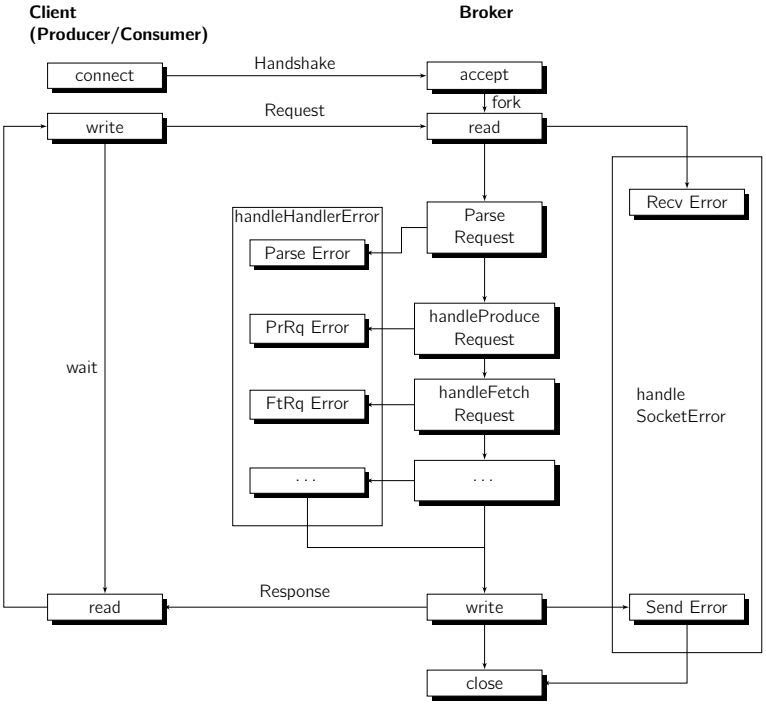


Figure 9.4.: Broker error handling concept in detail

Assume during the process of an incoming `ProduceRequest`, the log subsystem fails to write data to the disk and returns a log specific error. This error will not be handled directly. Instead, it is caught by the API layer which is now in charge to map this error to one of the defined `HandleError`. In this specific example, the request handler of the API layer takes the error response from the log system, maps it to a **String** (if it is not already mapped) as well as the offset of the failing message, and assigns this information to the type `PrWriteLogError` as the first and second argument.

The following part of this section will describe the functionalites of the *Socket Error Handler* as well as the *Request Error Handler* provided by the functions `handleSocketError` and `handleHandlerError`.

Socket Error Handler

The scope of handling a socket related error is very limited, due to the fact that the connection may be broken or the client will not receive any more data using the current connection. Given this, the current implementation only prints the occurred error on console:

```
1 handleSocketError :: (Socket, SockAddr) → SocketError → IO()
2 handleSocketError (sock, sockaddr) (SocketRecvError e) = do
3   putStrLn $ "[Socket Receive Error] " ++ e
4 handleSocketError (sock, sockaddr) (SocketSendError e) = do
5   putStrLn $ "[Socket Send Error] " ++ e
```

Listing 9.10: Handling error of type SocketError

While further logging of errors is out of scope within this thesis, the given architecture may very well be able to do so. At the point after the pattern matched–currently implements a **putStrLn**–one could inject an external logging service providing more meaningful information such as proposed in RFC5424[12].

Request Error Handler

The Apache Kafka Protocol defines error codes (see 8.1.4), which should be applied to a response message if the given failed for some reason. Thus, the `handleHandlerError` function is responsible to provide the related response message containing the appropriate error code. However, the current implementation does not suppport a response message caused by an error. Instead, it prints a notification on the broker side as shown in the code below:

```
1 handleHandlerError :: (Socket, SockAddr) → HandleError → IO()
2 handleHandlerError (s, a) (ParseRequestError e) = do
3     putStrLn $ (show "[ParseError]: ") ++ e
4 handleHandlerError (s, a) (PrWriteLogError o e) = do
5     putStrLn $ show "[WriteLogError on offset " ++ show o ++ "]: " ++
  show e
6 handleHandlerError (s, a) UnknownRqError = do
7     putStrLn $ show "[UnknownRqError]"
8 handleHandlerError (s, a) e = do
9     putStrLn $ show e
10    S.sendAll s $ C.pack $ show e
11    sClose s
12    putStrLn "***Host " ++ (show a) ++ "disconnected ***"
```

Listing 9.11: Handling error of type HandleError

[12] http://tools.ietf.org/html/rfc5424

9.5. Log Layer Subsystem

The subsystem of the log layer is responsible for the persistence of messages produced and consumed. This component plays a decisive role while handling a *fetchRequest* (9.3.2) as well as a *produceRequest* (9.3.2). Thus, it is used extensively in the API layer (9.3). In this section the applied concepts–which are adapted from Apache Kafka (6.2.1)–will be explained before a proper introduction to the components of the log subsystem is given. Afterwards follows a code explanation for the most important functionalities, where information about design decisions and potential threats are given.

9.5.1. Components

Even if the log subsystem lives within the broker environment, it is, from an architectural point of view, fully decoupled and relies only on the protocol library.

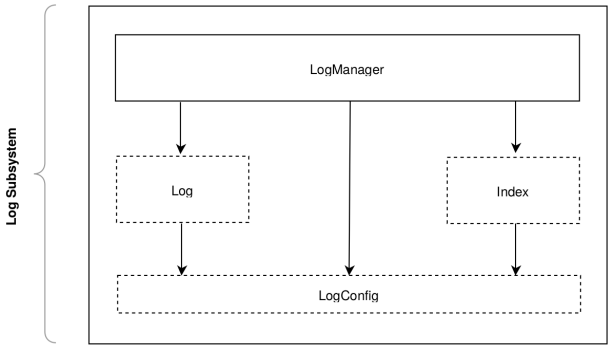


Figure 9.5.: Architecture of Log Subsystem

As illustrated in figure 9.5, the *LogManager* acts as an interface to any component that uses functionalities of the log, which in this case will be the API layer. The *LogManager* is then able to use exposed functionalities of the hidden modules *Log* and *Index* (displayed as dashed).

As for now, the functionalities *LogManager* provides creates a new log and appends or reads messages to/from an existing log. Further functionalities are provided but are still managed by the *Log* or *Index* module itself. Also exposed is the `State` type which is the in-memory representation of the available Logs, as discussed later in this section.

```
1 module HMB.Internal.LogManager
2 ( new
3 , append
4 , readLog
5 , State
6 ) where
```

The modules *Log* and *Index* contain functionalities with direct access on the filesystem. The separation leads back to the two provided files types log and index (see 9.5.2).

There are small amounts of commonly used functions in the *Log* as well in the *Index* module. The *LogConfig* seems to be the right location to place those intended to be imported with a qualified name such as simply `L`, standing for Log.

9.5.2. Storage Structure

The storage structure is exactly the same as the one defined for Apache Kafka (see 6.2.1). As for now, the root location of the log is not configurable and set to the folder named **./log** within the installation directory of the broker. Each folder represents a partition of a specific topic, whereas the name of the folder is a combination of the topic name and the partition number.

There are two types of files that reside within the folder specifying the topic and partition:

- **.log**: Containing a sequence of messages as binary data
- **.index**: Containing a sequence of index entries as binary data

Instead of holding only one storage file per topic-partition combination, the log can be segmented into multiple files due to reasons of lookup optimization. Apache Kafka starts segmenting a log after it reaches a configurable size whereas the default is one gigabyte. Therefore a topic-partition specific directory can contain multiple pairs of both file types. Each pair of file types hold the same name which stays for the base offset and can be considered as an unique identifier representing a 64 bit integer as a 20 character file name. As it was defined for Apache Kafka, the **base offset** is the offset of the first log entry stored in a file. For example if a log holds messages in where the first message has the offset, *5 :: Int64* the file name of this log will be *00000000000000000005.log* and the related index file *00000000000000000005.index*.

Using this naming convention and by viewing multiple log files, one can extract information about what range of messages reside in which log. This is a very efficient way to perform a lookup which is needed to append new log messages or archive old messages.

The following example shows a storage layout for topics *TopicA, TopicB and TopicC* with the partitions *0* and *1*. Partition 0 of topic A and B are already segmented into two combinations of an index and log file.

```
log
|
+--TopicA_0
   |--00000000000000000000.index
   |--00000000000000000000.log
   |--00000000000008650021.index
   |--00000000000008650021.log
+--TopicA_1
   |--00000000000000000000.index
   |--00000000000000000000.log
+--TopicB_0
   |--00000000000000000000.index
   |--00000000000000000000.log
   |--00000000000128655021.index
   |--00000000000128655021.log
+--TopicB_1
   |--00000000000000000000.index
   |--00000000000000000000.log
+--TopicC_0
   |--00000000000000000000.index
   |--00000000000000000000.log
```

9.5.3. Storage Format

Log File

The log file contains a sequence of messages, whereas the on-disk format of an entry is part of the reason why the Apache Kafka becomes valuable. In fact, the on-disk format is exactly the same as the format of the `MessageSet` (see 8.1.7) transmitted with a `ProduceRequest` (8.1.5). Thus, data does not have to be modified in any way and can directly be extracted and written to the file system.

Index File

Every segment of a log has its corresponding index represented as another file with the suffix *.index*. Whereas the log file contains the actual messages structured in a message format (see 8.1.7) and for each message within this file, the first 64 bits describe the incremented offset. Looking up this file for messages with a specific offset becomes very expensive since log files may grow in the range of gigabytes. And to be able to produce messages, the broker actually has to do such kind of lookups to determine the latest offset and be able to further increment incoming messages correctly. Also the fetch requests can use the index for finding the needed messages by given offset as fast as possible. This is why there is an index.

Basically, the index file contains simple entries with the following structure:

```
Relative Offset (4 Bytes)
Physical Position (4 Bytes)
```

The first field of an index entry, the relative offset, refers to a specific message in the corresponding log file. On the other hand, the second field represents the physical position of the referred message within the log file. Instead of storing the whole eight bytes of the message offset, the index entry only holds a relative offset to the base offset. This obviously reduces the size of the index. In addition, there is to note that not every message within the log is going to be indexed. By default, an index entry is created only after *4096 Bytes* of log data.

As per protocol specification, the request to fetch messages (see 8.1.5) holds the offset of the wanted message. Given this offset, one can determine the base offset and is discussed later in the log section (see 9.5.6). Assuming to have a valid base offset as well as the actual offset of the requested message, it is not a big deal to figure out the relative offset of this message:

```
Relative Offset = Given Offset - Base Offset
```

Given the relative offset, a lookup over the log can be processed. Remembering that the index is significantly smaller than the log and relies in memory, this lookup becomes reasonably fast. In fact, the file system will proceed a binary search on the index resulting in time complexity of O(log n), where n is the number of index entries.

A successful index lookup will then bring the physical position of the closest message of the requested message within the log, which is valuable. While the interval of index creation is fixed at a certain amount of bytes, a lookup of the actual log message, given the position of the closest indexed message, will result in time complexity of O(1).

9.5.4. Types

To make the code more readable and easier to manage, separate types are defined. These are only used in the log subsystem and are defined within the module *LogConfig*. The following table gives an overview of these types:

Type synonym	Parameter	Description
TopicStr	String	Parsed topic name as String
PartitionNr	Int	Parsed partition number as Int
RelativeOffset	Data.Word32	4 Byte offset relative to the BaseOffset
FileOffset	Data.Word32	4 Byte physical offset of a Log file
OffsetPosition	(RelativeOffset, FileOffset)	Tuple of relative and physical offset
BaseOffset	Int	8 Byte offset as Int

Table 9.1.: Types defined for log subsystem

9.5.5. In-memory Storage

Working with disk alone–in stateless fashion so to speak–is not only very difficult to manage (especially regarding multi- threading) but also extremely slow when dealing with thousands of requests in a short period of time (see). A more convenient and powerful approach needs to be introduced to handle a huge amount of data on a running broker system. Therefore, an intermediate storage in memory to optimize file accesses is demanded. An efficient implementation of a key-value data structure, namely Data.Map[13], seems to be suitable for both the log and index. Data.Map provides functionality that will come after all the needs of this subsystem, a time complexity for insertion, and searching of O(*log n*) scalability.

With this approach of using a data structure for the log as well as the index, it is more important than ever to lock appropriately in order to handle concurrent data access. Therefore the MVar[14] provides a flexible and powerful locking primitive. An MVar can be thought of as a box which is either empty or full. The `takeMVar` operation removes the value from a full MVar and returns it but waits (or blocks) if the MVar is currently empty. Symmetrically, the `putMVar` operation puts a value into the MVar but blocks if the MVar is already full. The MVar is a fundamental building block that generalizes many different communication and synchronization patterns. As for the log subsystem, the MVar is used as a *container for shared mutable state*. This is a common design pattern in Haskell when several threads need write access to some state where the state value represents an ordinary immutable Haskell data structure stored in an MVar—in this case a map of logs or indices. [27]

[13] https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html
[14] https://hackage.haskell.org/package/base-4.8.0.0/docs/Control-Concurrent-MVar.html

Log State

The `LogState` represents an MVar holding a map which uses a tuple of **String** and **Int**–representing the topic and partition combination–as key. The corresponding value is the actual log which is a list of `MessageSet`.

```
1 import qualified HMB.Internal.LogConfig as L
2 type Logs = Map.Map (L.TopicStr, L.PartitionNr) Log
3 newtype LogState = LogState (MVar Logs)
```

Listing 9.12: Definition of the log state (MVar)

Index State

Same as for the `LogState`, the `IndexState` represents an MVar containing a map of the same key. However, the value of this map holds a list of `OffsetPosition`, whose name is adapted form Apache Kafka. `OffsetPosition` represents a single index entry and therefore holds a tuple of the `RelativeIndex` and `PhysicalPosition`.

```
1 import qualified HMB.Internal.LogConfig as L
2 type Indices = Map.Map (L.TopicStr, L.PartitionNr) [OffsetPosition]
3 newtype IndexState = IndexState (MVar Indices)
```

Listing 9.13: Definition of the index state (MVar)

9.5.6. Determine Base Offset

This section gives an example of a common used function within the log subsystem, by explaining the function which determines the base offset.

An incoming request–which at this point already parsed–contains the topic as well as the partition number for the given set of messages. Thus, enough information is provided to identify the location–in this case the path–of the related log or index file. Because each log can be separated into multiple files, the thing that is still missing now is either in which file the messages should be appended (in case of a produce request) or from which file the messages should be read from (in case of a fetch request).

Getting existing base Offsets from File System

To get a list of all existing base offsets, several filters and string transformations need to be applied to the list of files in the topic-partition specific log directory.

First of all, a list of files has to be built. The library System.Directory[15] provides the function `getDirectoryContents` that takes a file path as a string and returns a list of all entires in the directory. This operation performs I/O, and thus the return type is an IO monadic value. *As described on hackage[16], there are several causes why this operation may fail but,*

[15] http://hackage.haskell.org/package/directory-1.2.2.1/docs/System-Directory.html

[16] http://hackage.haskell.org/package/directory-1.2.2.1/docs/System-Directory.html#v:getDirectoryContents

at this point, we do not provide proper error handling. The function `offsetFromFileName` can be mapped over a filtered list of strings, giving the list of all offsets within the directory. The filter function basically omits the files other than the ones ending with *".log"* as well as the root directories, typically *".", "..".*

```
1 getBaseOffsets :: (TopicStr, Int) → IO [BaseOffset]
2 getBaseOffsets (t, p) = do
3 dirs ← getDirectoryContents $ getLogFolder (t, p)
4 return $ map (offsetFromFileName) (filter (isLogFile) (filterRootDir dirs))
```

Listing 9.14: Determining all base offsets for given topic and partition

As hinted in the code above, a filter function is being used to get the list of all base offsets within the directory. The function *offsetFromFileName* extracts the offset as an *Int* from a *String*. As an example, *"00000000000000000005.log" :: String* will be transformed to *00000000000000000005 :: Int*:

```
1 offsetFromFileName :: [Char] → BaseOffset
2 offsetFromFileName = read ∘ reverse ∘ snd ∘ splitAt 4 ∘ reverse
```

Listing 9.15: Get numeric value (offset) from given string (filename)

Finally, the **Maybe** `Offset` helps to distinguish between the highest offset (**Nothing**) or the next smaller `BaseOffset` related to a provided **Just** `Offset`.

```
1 getLastBaseOffset :: (TopicStr, Int) → Maybe Offset → IO BaseOffset
2 getLastBaseOffset (t, p) o = do
3 bos ← getBaseOffsets (t, p)
4   case o of
5     Nothing → return (maxOffset bos)
6     Just o  → return (nextSmaller bos o)
```

Listing 9.16: Get highest base offset existing of given topic and partition

Getting right base Offset depending on Intent

In case of producing messages, the log segment wanted is the one named after the highest offset to be able to append new messages to the log. This is why one does not get around reading the content of this directory. Handling the case where no file exists leads to the creation of a new log with the offset of zero.

```
1 maxOffset :: [BaseOffset] → BaseOffset
2 maxOffset [] = 0
3 maxOffset [x] = x
4 maxOffset xs = maximum xs
```

Listing 9.17: Determining highest offset of given list

In case of receiving messages, the process is almost the same. The only difference is that the wanted file does not hold the highest offset but the one that hold the offset which is the next smaller one regarding the provided offset number within a fetch request. Let's assume one wants to fetch the message with offset *400*, yet there are two different pairs of log- and index files: *100* and *300*. The correct one would be *300* as the message must reside within this file.

```
1 nextSmaller :: [BaseOffset] → BaseOffset → BaseOffset
2 nextSmaller [] _ = 0
3 nextSmaller [x] _ = x
4 nextSmaller xs x = last $ filter (<x) $ sort xs
```

Listing 9.18: Determining offset which is next smaller regarding given offset

9.5.7. Append to a Log

Writing data of type `MessageSet` to a log is the fundamental process behind the scenes of handling a `produceRequest` (9.3.2). Before actually writing data to the file system, several steps come in between. These steps contribute significantly to the concept of Apache Kafka's Log system which are adapted in this implementation. This section describes the implementation details of the mentioned steps in the order of occurrence.

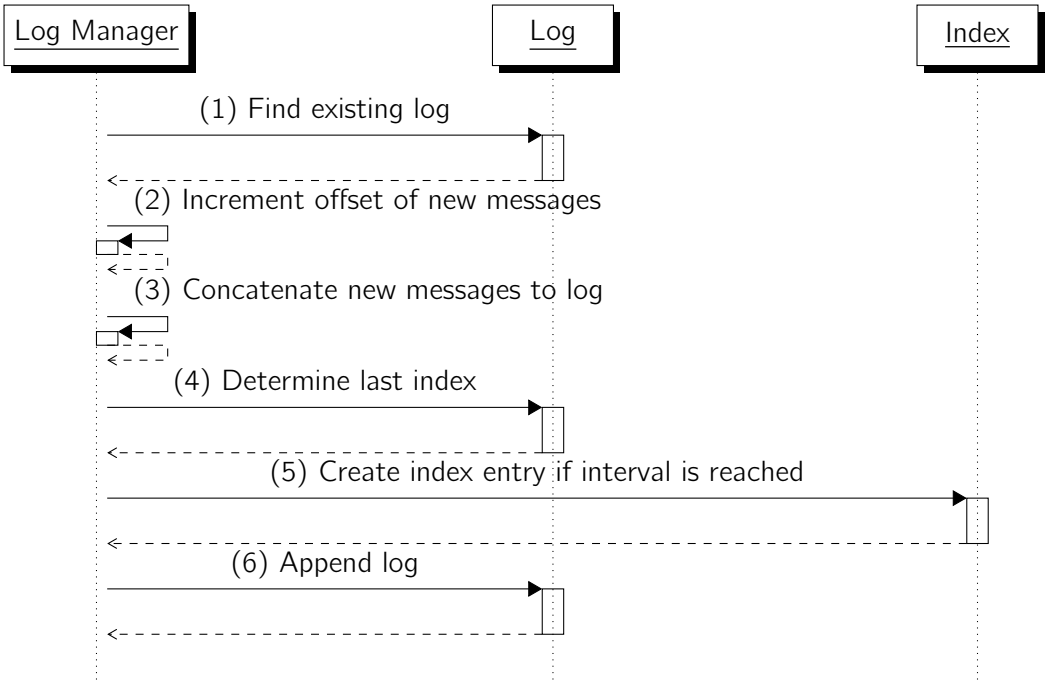


Figure 9.6.: The process of appending a message to the log

LogManager

The *LogManager* is responsible to append entries to a log, therefore exposing the function `append` which takes the tuple of both states as well as the topic and partition combination and the set of messages to append. As illustrated in figure 9.6, not only is the module *Log* involved in this process but also the module *Index*.

The concept behind the `append` function is that the *LogManager* takes the value of the MVar, computes log and index file specific functions in a pure fashion, and puts the value back in the MVar and after the append process is completed. During this process – between `takeMVar` and `putMVar` – the resource is locked. As for now, concurrent appending is not supported but may become possible in the future with this concept. The following listing describes the computations happening in between the mentioned unwrapping of MVar:

```
append :: (State, L.TopicStr, L.PartitionNr, Log) → IO ()
append ((Log.LogState ls, Index.IndexState is), t, p, ms) = do
  logs ← takeMVar ls
  let log = Log.find (t, p) logs -- (1)
  let llo = fromMaybe (-1) (Log.lastOffset log)
  let recvLog = Log.continueOffset (llo + 1) ms -- (2)
  let newLog = log ++ recvLog -- (3)

  indices ← takeMVar is
  let index = Index.lastOrNull (Index.find (t, p) indices) -- (4)
  bo ← L.getBaseOffset (t, p) Nothing
  let lastIndexedOffset = fromIntegral (fst index) + (fromIntegral bo)
  if Index.isInterval (Log.sizeRange (Just lastIndexedOffset) Nothing newLog) -- (5)
     then do
        syncedIndices ← Index.append indices (t, p) newLog (Log.size newLog)
        putMVar is syncedIndices
     else do
        putMVar is indices
  let newLogs = Map.insert (t, p) newLog logs
  syncedLogs ← Log.append (t, p) newLogs -- (6)
  putMVar ls syncedLogs
```

Listing 9.19: Append function exposed by LogManager

(1) The log relating to the given topic and partition combination is searched within all existing and not to the disk persisted logs residing in the in-memory data structure (9.5.5). Except for the first message topic and partition combination, there will always be at least one element within this map. The remaining element in this list is required to later continue the offset number for the incoming message sets.

(2) Having the very last entry in a log allows to further increment the offset over the set of messages which need to be appended.

(3) The received messages are append to the existing log.

(4) At this point, it may now be possible that the updated log reached the size of the defined interval for indexing a new log entry. The first step in order to determine this is to get the offset of the last index entry. Keep in mind that the index only holds the relative offset; the base offset has to be added.

(5) The module *Log* provides the function `sizeRange` that takes a start and end offset and calculates the size in bytes of this range of messages. If this size reaches the index interval (4096 bytes), an index entry will be created.

(6) Finally, the updated loh can be written to the in-memory state and eventually to disk. Writing to disk is hidden in the `append` function of the module *Log* as it batches messages and only writes if the total in-memory size of 500 messages is reached.

Log Module

Initiated by the *LogManager*, the `append` function of the *Log* module takes not only the in-memory map of the specific log but also the updated log. As topic and partition are required anyways in this function, working with the entire collection is easier to handle on the *LogManager* side.

```
append :: (L.TopicStr, L.PartitionNr) → Logs → IO Logs
append (t, p) logs = do
  let logToSync = find (t, p) logs
  if isFlushInterval logToSync
      then do
          write (t, p, logToSync)
          let keepLast = [last logToSync]
          return (Map.insert (t, p) keepLast logs)
      else return logs
```

Listing 9.20: Append messages to log

The flush interval describes the length of a given log and returns true if the amount of messages reached the limit of 500, which is the default value of Apache Kafka.

```
isFlushInterval :: Log → Bool
isFlushInterval log = 500 ≤ length log
```

Listing 9.21: Check if enough messages are given for flush to disk

If the flush interval is not reached yet, the entire collection is returned. This is the step what makes writing to disk fast since messages are not written to disk for each request but instead are batched until this limit is reached.

If the interval is reached, the actual process of writing to disk can begin. The batched in-memory log will be encoded and eventually appended to the log file. The `withFile` function from System.IO [17] opens the file and passes the resulting handle to the computation act. The handle will be closed on exit from `withFile`.

```
write :: (L.TopicStr, Int, Log) → IO ()
write (t, p, ms) = do
  bo ← getBaseOffset (t, p) Nothing
  let logPath = L.getPath (L.logFolder t p) (L.logFile bo)
  withFile logPath AppendMode $ λhdl → BL.hPut hdl $ encode ms
```

Listing 9.22: Write message to file with AppendMode

Index Module

Appending an index entry requires the relative offset of the last message from the set of messages given by the *LogManager*. Therefore, the base offset will be subtracted from the offset of the last message. The second value of the tuple is the physical offset. The physical offset basically represents the current size of the log file, including the size of the upcoming log entries persisted later during this request cycle. Thus, after the messages are persisted, adding both sizes will result in the correct physical offset.

[17] https://hackage.haskell.org/package/base-4.2.0.1/docs/System-IO.html

```
append :: Indices → (L.TopicStr, L.PartitionNr) → Log → Int64 →
IO Indices
append idx (t, p) ms logSize = do
  let old = find (t, p) idx
  bo ← L.getBaseOffset (t, p) Nothing
  let path = L.getPath (L.logFolder t p) (L.indexFile bo)
  fs ← getFileSize path
  let new = pack (fromIntegral (msOffset (last ms)) - bo
                                  , fs + (fromIntegral logSize))
  let newIndex = old ++ [new]
  return (Map.insert (t, p) newIndex idx)
```

Listing 9.23: Appending new entry to index

At the moment, only in-memory persistence is supported for the log. On-disk persistence is fundamentally important to recover the index if the broker restarts accidentally or not.

As for now, index append remains as a major performance issue. Each time an index entry is being created, `L.getBaseOffset` results in an additional I/O where a lookup of the directory structure is being made as well as string transformation is proceeded. Besides this, the file size of the log file also has to be determined. Since everything is single-threaded, this becomes even worse. More details and approaches to solve this can be found in chapter 10.

9.5.8. Read from a Log

Reading from a log means that consumers have subscribed to a topic (including partition) and will send *fetchRequests* to consume data from the broker. This data was originally generated by producer clients and, as described in , is processed by the broker and persisted on disk storage.

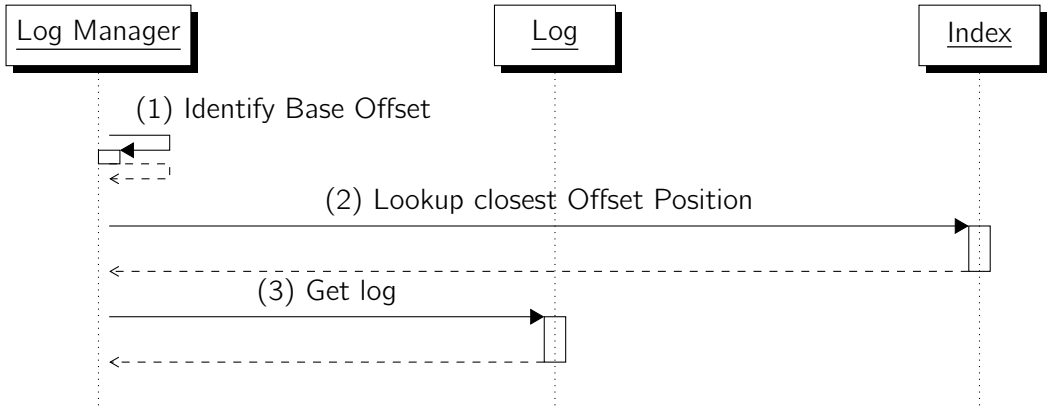


Figure 9.7.: The process of reading a log

LogManager

Analog to appending, the log manager provides the interface to read from a log. The function `readLog` takes the tuple of topic and partition as well as the offset from with to start to fetch. A consumer expects to get the message which has a offset equal or larger than the fetched offset. If there is no message in the log with given offset or higher, the consumer gets a empty list. The ability to allow the consumer to limit the requested size of messages in bytes is not implemented yet.

```
1 readLog :: (Index.IndexState, L.TopicStr, L.PartitionNr) → Offset →
  IO Log
2 readLog (Index.IndexState is, t, p) o = do
3 bo ← Log.getBaseOffset (t, p) (Just o) -- (1)
4   idx ← takeMVar is
5   let index = Index.find (t, p) idx
6   op ← Index.lookup index bo o -- (2)
7   putMVar is idx
8   log ← Log.lookup (t, p) bo op o  -- (3)
9   return log
```

The concept behind `readLog` can be done with the following three steps:

(1) The base offset has to be identified in order to know which log file the wanted message is residing in and from which index file the related offset position can be taken.

(2) Next step leads to the lookup of the index by providing the offset of the message the consumer is looking for. This will return a tuple of base offset and physical position of the closest indexed message.

(3) Having the physical position of the message, which is the closest to the message that should be consumed, helps while looking up the log. In fact, the range to look for can be limited enormously. Starting position for the lookup of the log file can be set directly to the physical position of the indexed message.

Index Module

Given the highest offset for a specific topic name and partition (base offset), the next step determines the offset entry which is the closest to the effectively wanted offset. Therefore, the map containing the topic and partition related index is searched.

```
1 lookup :: [OffsetPosition] → BaseOffset → Offset → IO OffsetPosition
2 lookup index bo o = do
3   let relOffset = o - fromIntegral(bo)
4   return (findOrNextSmaller relOffset index)
```

Log Module

In order to increase I/O performance, the index file will be mapped into memory first using mmap[18]. The resulting memory-mapped file implements demand paging, which is an operation that copies a disk page into physical memory only if an attempt is made to access

[18] http://man7.org/linux/man-pages/man2/mmap.2.html

it or the page is not already in memory. This comes useful in terms of Haskell's laziness such as that a lazy ByteString can be taken as a result of this operation, and thus memory consumption can be considered as moderate. But even more importantly, the amount of I/O operations will be decreased drastically for multiple lookups on the index file.

Haskell provides the System.IO.MMap library which is an interface to mmap(2) system call under POSIX. Fortunately, the mmap wrapper function *mmapFileByteStringLazy* takes a *Maybe (Int64, Int64)* to specify the range of the file to be mapped. The range is defined by the offset, which is the beginning byte of the file region and its size tells the mapping length.

After getting the bytestring from the range of the physical position given by the index to the end of the log it can be encoded to a list of messages. Because the index entry could refer to a message behind the actual demanded offset the encoded list finally needs to filter out messages with a offset smaller than the requested.

```
1 lookup :: (L.TopicStr, Int) → L.BaseOffset → L.OffsetPosition →
  Offset → IO Log
2 lookup (t, p) bo (_, phy) o = do
3   let path = L.getPath (L.logFolder t p) (L.logFile bo)
4   fs ← getFileSize path
5   bs ← mmapFileByteStringLazy path $
6           Just (fromIntegral phy, (fromIntegral (fs) - fromIntegral phy))
7   let log = runGet decodeLog bs
8   return $ filterByOffset o log
```

10. Conclusion

10.1. Results

10.1.1. Technology Research

The first result of the thesis is the documentation of the technology research, which results in gathered knowledge that is further required for this thesis. It gives an insight into messaging fundamentals and takes a closer look at Apache Kafka and related topics.

10.1.2. Protocol Implementation

A fundamental result of the thesis is the implementation of the Apache Kafka protocol in Haskell. The design decision of separating protocol related code from the broker implementation lead to an isolated product which can now be used as a library for different projects. The code is provided through an open sourced repository[1] for further development.

A complete implementation of the Apache Kafka Protocol would go beyond the scope of the thesis, as the focus lies not only in implementing the protocol but also to provide broker functionality. Thus, most important is the ability to produce and fetch messages. In order to provide compatibility for Apache Kafka clients, the Metadata API is also required because it is the first part of the process in producing or consuming messages where it will request information about the broker and provided topics. The following list gives an overview of which part of the protocol is implemented and what remains open:

- ✓ Metadata API
- ✓ Produce API
- ✓ Fetch API
- ✗ Offset API
- ✗ Offset Commit/Fetch API

[1] https://github.com/hmb-ba/protocol

10.1.3. Haskell Message Broker

The main approach of this work is the implementation of a message broker in Haskell. The resulting application provides a server with basic functionality in handling requests via network and persisting messages. The broker is fully Kafka compatible since it is based on the protocol implementation. In comparison to the original Apache Kafka, there are, of course, a lot of features missing in order to offer the same functionality. Furthermore, as the scope of the thesis lies on a single broker system, there is not yet support for broker replication. Nevertheless, the provided Haskell message broker is a prototype which shows feasibility of developing a similar system in a fully-fledged functional program language like Haskell. The following list gives an overview of what features the broker system already provides and what remains open in the particular area:

Produce API:

- ✓ Publishing messages to a specific topic and partition
- ✓ Publishing to multiple topics and partitions per request
- ✓ Supporting batched messages
- ✗ Configurable Acknowledgements and Timeout

Fetch API:

- ✓ Consuming messages of a specific topic and partition depending on given offset
- ✗ Supporting consuming from multiple topics and partitions per request
- ✗ Supporting configurable min and max bytes for fetched data
- ✗ Supporting consumer groups data is available

Metadata API:

- ✓ Fetching available topic names from broker
- ✓ Include broker information
- ✗ Include partition status
- ✗ Include replication information

Persistence (Log):

- ✓ Persist messages in topic-partition specific log structure
- ✓ Read messages from log, optimized with index file
- ✓ Provide LogManager for handling files and directories structure
- ✗ Write Index to disk and rebuild the memory after the broker is restarted
- ✗ Data retention, where old data is discarded after a fixed period of time or Kafka's log compaction is applied.

10.2. Evaluation

After highlighting the implemented features above, the question about performance still remains. Is the provided prototype potentially faster than the original Apache Kafka? Does the prototype near the same throughput? Or are we still far away from any approximation to the impressive performance of the reference system? This chapter describes the environment and tools in which the systems were tested. It also gives insight in early stress tests which were very useful in finding network related bugs. Additional benchmarks are defined to demonstrate the performance of the final version of the Haskell message broker. For comparison, the original Apache Kafka system is involved in the benchmarks by running equivalent tests on the same hardware.

10.2.1. Test Setup

All of the tests and benchmarks in this chapter are performed with following setup:

```
Broker Server:
  Ubuntu Server 14.04.2 LTS (64 bit)
  Intel(R) Xeon(R) CPU E3-1245 v3 @ 3.40GHz
  One 7200 RPM SATA drive
  16GB of RAM
  1Gb Ethernet

Client:
  Ubuntu 14.04.2 LTS (64 bit)
  Intel(R) Core(TM) i7-2720QM CPU @ 2.20GHz
  One SSD SATA drive
  16GB of RAM
  1Gb Ethernet
```

10.2.2. HMB Performance Producer

The HMB performance producer is a simple console application used for stress tests in the broker system. It can be used to generate messages of a given size and batch them into a single request. It then repeatedly calls the client library function `sendRequest` to pack, encode, and send the request. The following code shows the simplified main function:

```
1  import qualified System.Entropy as E
2
3  main = do
4      -- Socket setup
5      -- ◦ ...
6      -- Get numberOfBytes and batchSize as variable input
7      -- ◦ ...
8
9      randBytes ← E.getEntropy numberOfBytes
10     let batch = [randBytes | x ← [1..batchSize]]
11
12     replicateM_ 1000000 (sendRequest sock req)
```

```
13      putStrLn "done produce"
14      return ()
```

10.2.3. Kafka Performance Producer

The Kafka Performance Producer[2] is a official product of Apache Kafka that systematically tests their broker system. It is a wrapper around the original Kafka producer client for producing as many messages as possible. It provides many configuration options[3] for specific benchmarks.

The following listing shows the setup for the benchmarks of the thesis. Values in brackets [ ] are placeholders for the relevant and variable parameters for the benchmarks. The used version of Kafka is 2.10-0.8.2.0.

```
Start Zookeeper
>bin/zookeeper-server-start.sh config/zookeeper.properties

Start Kafka Node
> bin/kafka-server-start.sh config/server.properties

Create test topic [TopicName] on broker server with [IP],
one partition, no replication:
>bin/kafka-topics.sh --zookeeper [IP]:2181
    --create --topic [TopicName] --partitions 1 --replication-factor 1

Start producing benchmark:
> bin/kafka-run-class.sh org.apache.kafka.clients.tools.ProducerPerformance
    [TopiName] [NumberOfMessages] [MessageSize] -1 acks=1
    bootstrap.servers=[IP]:[Port] buffer.memory=67108864 batch.size=[BatchSize]
```

10.2.4. Exposed Issues

An early version of the broker provided a console client which produced messages per console input. This setup demonstrated that encoding/decoding of a request worked. Also, the different broker layers have done their job well. Later in the construction phase of the broker, tests were introduced that repeatedly sent a large amount of requests in a short period of time. This kind of stress test is very useful to uncover network related bugs and performance lacks. We kept doing so until the end of the implementation phase.

When dealing with thousands of requests per second, a function with high complexity can cause terrible bottlenecks. Another method to analyze Haskell code for potentially slow functions is time and allocation profiling provided by GHC.

The following table lists some of the major issues that were exposed:

[2] https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/tools/ProducerPerformance.java

[3] http://kafka.apache.org/documentation.html#newproducerconfigs

Exposed Issue	Cause	Solution
Dramatically slowdown after short period of time. **[Solved]**	TCP Zero Window - Socket receive buffer of broker full	Broker application was slow in receiving, parsing, and handling incoming requests. Threading concept needs to be optimized (see 9.1). The process of parsing an incoming request needs to be swapped out to another thread, so that the network receive thread can reduce the socket buffer fast enough.
Parsing from socket error "not enough bytes" when testing over network interface instead of loopback. **[Solved]**	The `len` argument to the `recv` system call is merely the upper bound on the received length (e.g. the size of the target buffer); the call may produce less data than asked for.	Introduced function which checks if the exact amount of bytes is really read. If not, the `recv` call is repeated until all requested bytes are present (see details in 9.2.2).
Throughput drop when producing a larger request with nested list. **[Solved]**	Issue in encoding a request called `runPut` on each sub element and appended to a whole.	No appending of parsed sub elements. Restructure encode module by running `runPut` at latest point in time. This resulted in factor 2 of performance in building requests (see 8.2.2).
Slow performance in persisting messages to the log (file system). **[Solved]**	No state and batching at broker side. Each incoming message ended in multiple file accesses which slowed down the whole process.	Messages need to be summed up to larger chunks of bytes. After a defined period of time or amount of bytes, multiple messages need to be flushed to disk at once (see details of different approaches in 9.5).
Slow index search. **[Open]**	Base offset has to be determined for each lookup. As for now this causes disk I/O which is obviously an overhead.	The directory and its content should be read from disk just once and from then on work with states (MVar) only. The state has to be passed forth and back between the API and Log layer for each request but omits I/O by having the base offset in memory all the time.
Slow index insertion causes slowdown of the broker application. **[Open]**	As for now, the file size of the log which needs to be stored within an index entry causes I/O.	A more intuitive and significantly faster solution would be to maintain the log file size within the `LogState` (in-memory). Thus, the file size would have to be increased by hand when appending a new log entry.

Table 10.1.: Uncovered issues in early stress tests

10.2.5. Benchmark: Effect of Message Batch Size

The goal of this benchmark is to analyze the network throughput between producer client and broker system. The focus lies on the effect of changing the batch size at producer client to the resulting transmission of bytes per second. The batch size determines the amount of bytes in which messages are packed together and sent to the broker as a single request for efficiency. Increasing the batch size requires more memory. The size of a single message is fixed to 100 bytes because it is generally the more interesting case for a messaging system. It is much easier to get good throughput in MB/sec if the messages are large (as shown in second benchmark, see 10.2.6) but harder to get good throughput when the messages are small, as the overhead of processing each message dominates.

Calculation batch size in bytes:

```
Messages Batched * (Message Size + Message Overhead) + Request Overhead
```

Conditions

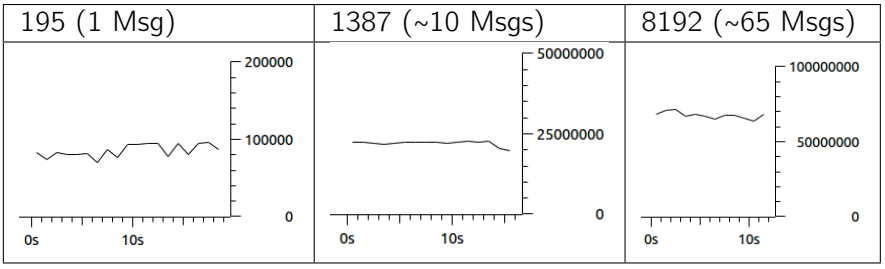
Message Size	100 Bytes
Batch Size [B]	195 | 1387 | 8192 | 16384 (Kafka default) | 56769 Bytes
Tested Clients	• Kafka Performance Producer 2.10-0.8.2.0, single threaded • HMB Producer, single threaded
Tested Brokers	• Kafka Broker 2.10-0.8.2.0, no replication, one partition • HMB Broker, no replication, one partition
Measurement	Resulting TCP throughput over a period of 10 seconds analyzed with Wireshark. Throughput includes message + request overhead
Scenarios	Producing as much messages as possible from client (left) to broker (right). The size of messages batched together varies in defined steps. 1. Kafka Performance Producer → Kafka Broker 2. Kafka Performance Producer → HMB Broker 3. HMB Producer → Kafka Broker 4. HMB Producer → HMB Broker

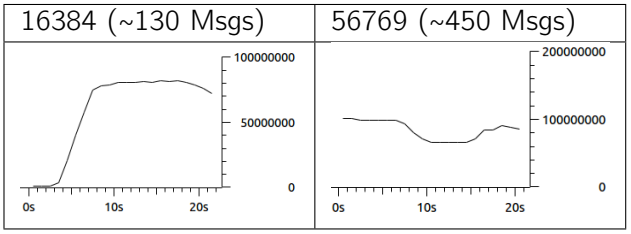
Table 10.2.: Benchmark conditions "Effect of Producer Batch size"

Results

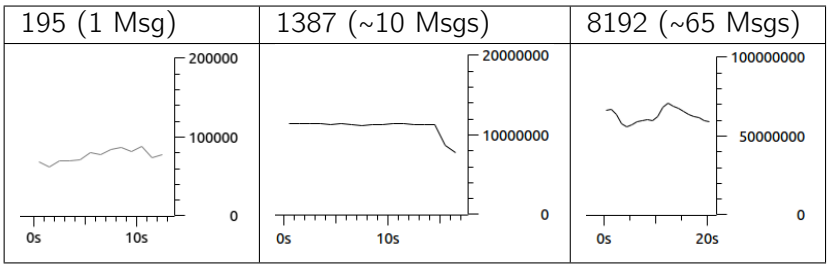
The results are given below for each specific scenario. The graph shows the TCP throughput in bytes in period of 10 to 20 seconds.

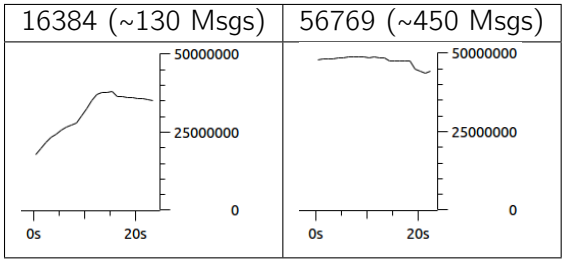
Scenario 1, Kafka → Kafka:



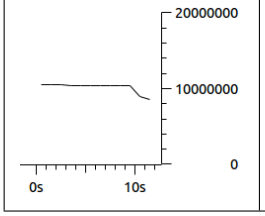
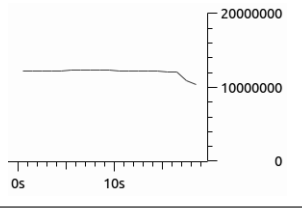
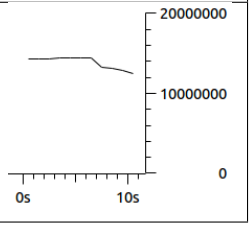


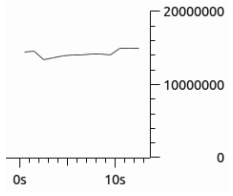
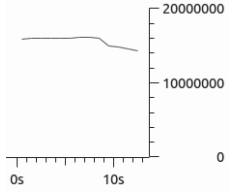
Scenario 2, Kafka → HMB:



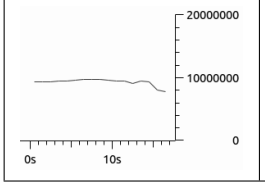
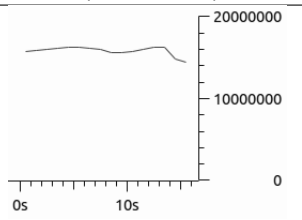
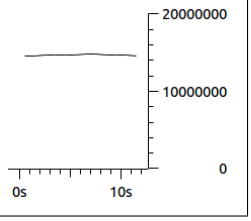


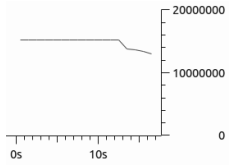
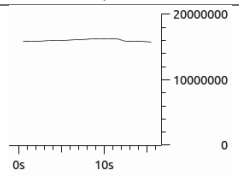
Scenario 3, HMB → Kafka:

195 (1 Msg)	1387 (~10 Msgs)	8192 (~65 Msgs)
		

16384 (~130 Msgs)	56769 (~450 Msgs)
	

Scenario 4, HMB → HMB:

195 (1 Msg)	1387 (~10 Msgs)	8192 (~65 Msgs)
		

16384 (~130 Msgs)	56769 (~450 Msgs)
	

Conclusion

Benchmark Variable	Avg. Throughput [MB/s]			
Batch Size [Byte]	Scenario 1	Scenario 2	Scenario 3	Scenario 4
195 (1 Msg)	9	8	10	9
1387 (~10 Msgs)	24	12	11	16
8192 (~65 Msgs)	68	54	13	15
16384 (~130 Msgs)	80	32	14	16
56769 (~450 Msgs)	98	48	16	17

Table 10.3.: Results of benchmark "Effect of Producer Batch Size"

The result of the benchmark demonstrates the positive effect to throughput by increasing the producing batch size. The original Kafka setup (scenario 1) shows a significant rise by increasing the batch size. However, the HMB setup (scenario 4) cannot use batching to full capacity. Obviously, this benchmark exposes the HMB producer to a major bottleneck. Using the Kafka producer to the HMB broker (scenario 2) gives better results instead of using the HMB producer. Scenario 3 confirms this suspicion. The handling of requests on the HMB broker is quite efficient, but there is still a demand for further optimizations to get the same performance of Kafka.

Consequences

During the benchmark, it was conspicuous that the HMB broker is not using the full capacity of the CPU. While Kafka obviously uses multi-threading, most of the load weighs on a single thread for HMB. This is why there is only one API handler thread running at the time, which is definitely a point to optimize in the future.

Another point in demand with optimization is the producer which manifests a bottleneck in messages of 100 bytes (when the message size gets bigger, the result is much better, see 10.2.5). Because the provided HMB producer is quite simple (actually just built to test), another task for the future is to work off details for optimized producer clients which would definitely lead to better results.

10.2.6. Benchmark: Effect of Message size

The goal of this benchmark is to analyze the network throughput between producer client and a broker system. The focus lies on the effect of changing the size of a single message to the resulting transmission of bytes per second. Persisting performance on the broker is not considered in this benchmark.

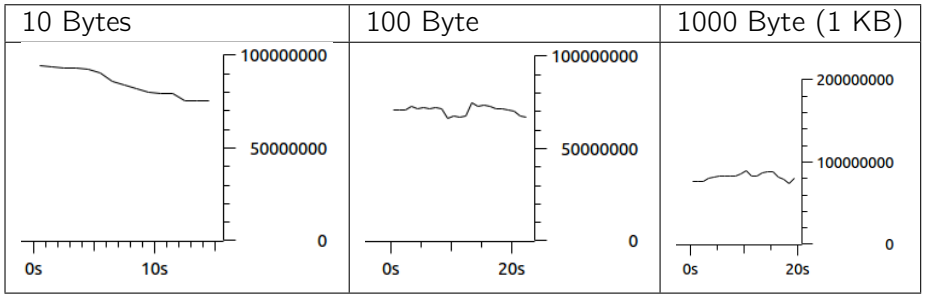
Conditions

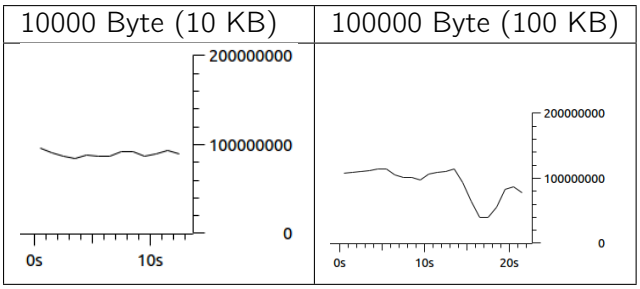
Message Size	10 | 100 | 1000 | 10000 | 100000 Bytes
Batch Size	12800 Bytes
Tested Clients	• Kafka Performance Producer 2.10-0.8.2.0, single threaded • HMB Producer, single threaded
Tested Brokers	• Kafka Broker 2.10-0.8.2.0, no replication, one partition • HMB Broker, no replication, one partition
Measurement	Resulting TCP throughput over a period of 10 seconds analyzed with Wireshark. Throughput includes message + request overhead.
Scenarios	Producing as much messages as possible from client (left) to broker (right). The size of a messages varies in defined steps. The batch size thereby is a fixed value. 1. Kafka Performance Producer → Kafka Broker 2. Kafka Performance Producer → HMB Broker 3. HMB Produer → Kafka Broker 4. HMB Produder → HMB Broker

Table 10.4.: Benchmark conditions "Effect of Message size"

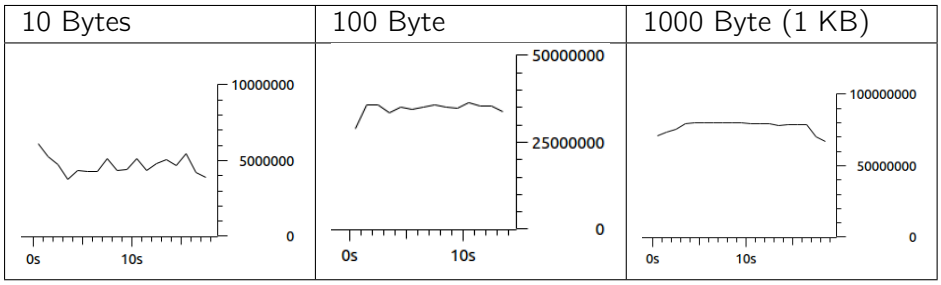
Results

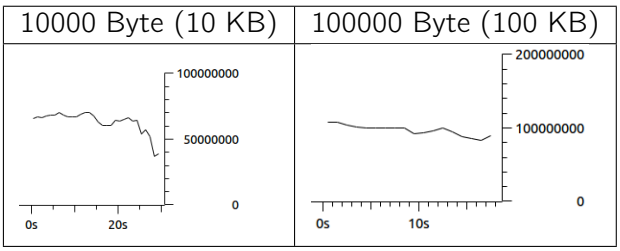
Scenario 1, Kafka → Kafka:



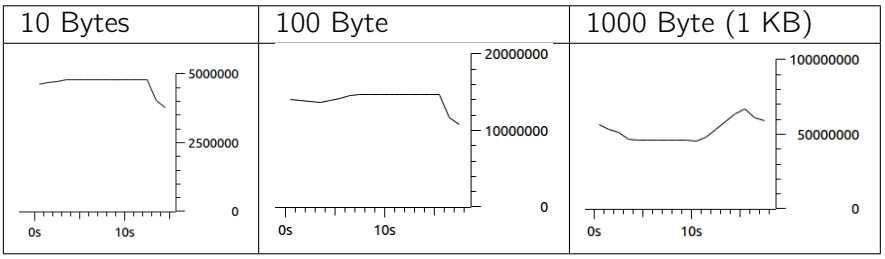


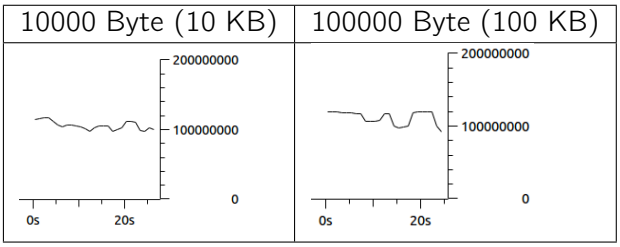
Scenario 2, Kafka → HMB:



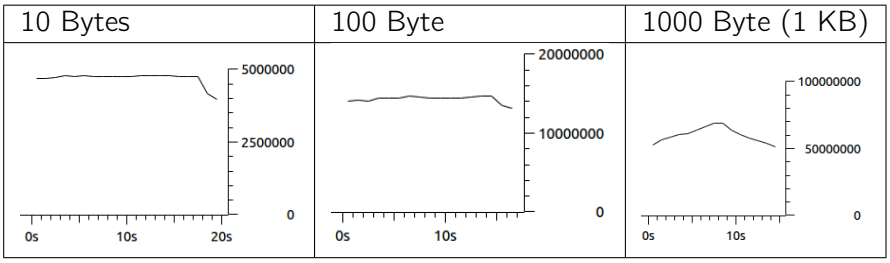


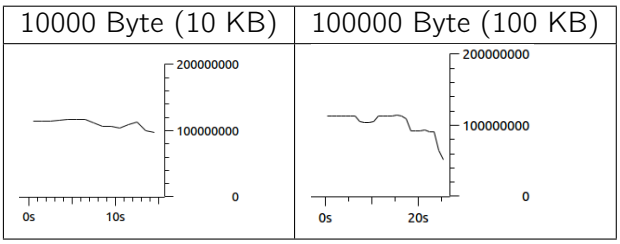
Scenario 3, HMB → Kafka:





Scenario 4, HMB → HMB:





Conclusion

Benchmark Variable	Avg. Throughput [MB/s]			
Message Size	Scenario 1	Scenario 2	Scenario 3	Scenario 4
10 B	85	50	48	48
100 B	70	35	15	15
1000 B (1 KB)	80	75	50	55
10000 B (10 KB)	95	75	100	100
100000 B (100 KB)	105	100	105	100

Table 10.5.: Results of benchmark "Effect of Message size"

The throughput increases as the message size gets bigger. First of all, why are tests with 10 bytes a message significantly faster than those with 100 bytes? It needs to be considered that the TCP throughput measurement takes bytes of the messages as well as the message and request overhead which is produced for transmitting the request. Each message has an overhead of 28 bytes which, in this case, is 280 percent of the actual message size. Therefore, the tests with 10 bytes are not really meaningful at the moment. As the message size gets larger, the overhead can be disregarded.

The benchmark shows that the HMB broker and producer can deal with large message sizes quite well! The resulting throughput of 10 and 100 KB messages is nearly the same as the Kafka setup. As already detected in the first benchmark (see 10.2.5), the HMB producer has its troubles with messages around 100 bytes.

Consequences

This benchmark has uncovered the responder thread as a bottleneck with larger message sizes because the socket buffer overflows. The test results are made without sending responses from the broker. At this point, an optimization in the network layer is needed so that the socket buffers never get too large.

Furthermore, the consequences defined in first benchmark also apply in this case.

10.3. Experiences with Haskell

In this section we want to give a statement about the suitability of Haskell as programming language regarding to the requirements of this work as well as our own experiences during the thesis.

10.3.1. Assets

Haskell type system. Regarding to a protocol implementation one can take advantage of the Haskell type system. The given grammar in the case of the Apache Kafka protocol can be mapped with types very well. Instead of having a big mesh of interfaces and classes as it would be in object oriented languages, the mapping in Haskell will get to a compact and good readable code where changes can be made easily. The power of Haskell types also manifests when compiling a given code. The type checker subsequently tell any mismatches at exactly point in the code. We felt very comfortable with this strong and efficient ability of the GHC and refactoring was getting much easier. When we started to implement the first prototype for the thesis we intuitively started with the type base. Retrospectively it was the right decision because the advantages mentioned above led to working code quite fast.

Compactness of code. We had come to love with the functional programming concepts Haskell provides, especially *pattern matching*, *list comprehensions*, *guards*, *maps*, *filters* or *lambdas*. Those functionalities allow programmers to write proper, elegant and compact code which we think is definitely a benefit for every work. Concepts like *lamdba* are already present in many other languages (not only functional ones). We are certain that we can use the learned abilities to write proper code in any futur project.

Sources. Simply searching answers to a specific Haskell problem in the internet rarely leads to a specific answer. Although we think that the provided information based from the Haskell community are still very good. One does just have to know how use them. First of all, Hoogle[4], as Haskell API search engine was very useful to search the standard libraries by any functions. A very helpful feature of Hoogle is searching by type signature (e.g. `(a→b) → [a] → [b]`) which allows to find functions without any given name. Another tool is Hackage[5] which provides access to any open sourced cabal package from the Haskell community. Each package includes detailed description about provided modules with their functions and types and how to use it. When we came at the point where we needed more information about a specific library or a problem, we acquired the community directly via the Haskell Subreddit[6] where we made very good experiences.

[4] https://www.haskell.org/hoogle/
[5] https://hackage.haskell.org/
[6] https://www.reddit.com/r/haskell/

10.3.2. Difficulties

Cabal Hell. To properly separate the components of protocol and broker implementation we used Haskell Cabal[7]. It provides a common system for building a packaging Haskell libraries and programs. Defining our components as single cabal packages leads to an independent protocol implementation package which could be used in any other project. So far so good. Unfortunately the well-known expression of *Cabal Hell* manifested its right. The fact that reinstalling a package with `cabal install` can break existing packages on a system was very relevant to our project because we obviously worked on two dependent packages simultaneously. We needed to reinstall packages all the time. Fortunately cabal provides the `sandbox` functionality, which detects modifications on local depended packages and reinstalls them automatically. Although this worked quite well, we still had a lot of cases in which our packages broke anyway. Then we had to reinitialize the sandbox setup and reinstall and compile all packages many times. Problems getting really serious when we tried to reinstall any cabal package with profiling features. Best experience we made by installing the *cabal* command line tool according to the official Github readme[8].

Lazy Evaluation. Programming a server application with lots of I/O parts often led us to think in sequential kind of manner, whereas we expected that every line of code gets called exactly at the point where it appears in code. That is the false thinking. Actually due to Haskell's lazy evaluation ability, a function is not called until the value is really needed (e.g. for print out to console). As beginners in the functional paradigm this was a kind of a difficulty to deal with. But thinking in terms of streams and laziness allows programmers to write elegant and compact code which would be awkward in eager language.

[7] https://www.haskell.org/cabal/

[8] https://github.com/haskell/cabal/tree/master/cabal-install

10.4. Outlook

What are the next steps? As described in section 10.1, the resulting broker implementation is not a finalized product–it is a prototype. It shows that it is feasible to develop an Apache Kafka-like broker system written in Haskell. An extendible and scalable implementation base and architecture is provided. The provided benchmarks (10.2.5 and 10.2.6) demonstrate good performance but also uncover some major bottlenecks. We are certain that, with further work, one could build the current version to an extraordinary broker system.

Summarized, the next tasks to do with the prototype involve:

1. Facing uncovered bottlenecks (see benchmarks)
2. Implementation and handling of remaining APIs
3. Extending the log subsystem
4. Work off more details for Client API
5. Further optimizations

After bringing the broker server to a stable and nearly feature complete application, the next step is introducing broker replication. This task is probably predestined for another thesis with a goal in analyzing distributed replication in detail and working out ZooKeeper integration.

The Haskell community is very vital and active. We experienced that at the ZuriHac 2015[9] (Google Zurich, 29.05.15) and in the HaskellerZ Meetup[10] (ETH Zurich, 30.04.2015) we participated in during the thesis. The results of this work will be published open source through the GitHub repositories[11]. The goal is to find contributors for further development. For this reason, we will also present this work in a later Meetup in summer 2015. As for now, the highlight remains the protocol implementation that already has been praised by the Haskell community, where contributors helped uncovering minor issues.

[9] https://wiki.haskell.org/ZuriHac2015
[10] http://www.meetup.com/de/HaskellerZ/
[11] https://github.com/hmb-ba

Part IV.

Appendix

A. Listings

B. Bibliography

[1] Activemq. http://activemq.apache.org/. last visited: 20.03.2015.

[2] Amazon kinesis faq. http://aws.amazon.com/kinesis/faqs/. last visited 25.03.2015.

[3] Amazon kinesis faq. http://aws.amazon.com/sqs/faqs/. last visited 25.03.2015.

[4] Amazon kinesis product details. https://aws.amazon.com/kinesis/details/. last visited 25.03.2015.

[5] Amazon sqs product details. http://aws.amazon.com/sqs/details/. last visited 25.03.2015.

[6] Apache flume - user guide. http://archive.cloudera.com/cdh/3/flume/UserGuide/index.html. last visited: 20.03.2015.

[7] Apache foundation. http://www.apache.org. last vistied: 19.03.2015.

[8] Apache kafka. http://kafka.apache.org/. last visited: 01.06.2015.

[9] Apache kafka: Next generation distributed messaging system. http://www.infoq.com/articles/apache-kafka. last vistied: 19.03.2015.

[10] Apache kafkaa wiki - replication. https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication. [Online].

[11] Apache zookeeper. https://zookeeper.apache.org/. last visited 25.03.2015.

[12] The eight fallacies of distributed conputations.

[13] Fallacies of distributed computing explained.

[14] A guide to the kafka protocol.

[15] Implementing pub/sub based on aws technologies.

[16] Linkedin inc. https://linkedin.com. last visited: 03.05.2015.

[17] Making sense of it all, lambda architecture.

[18] Rabbitmq. http://www.rabbitmq.com. last visited: 20.03.2015.

[19] C Bange, T Grosser, and N Janoschek. Big data survey europe: Usage, technology and budgets in european best practice companies. *Würzburg: BARC Institute*, 2013.

[20] Peter Sommerlad Buchmann Frank; Regine Meunier, Hans Rohnert and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley and Sons Ltd, 1 edition, 8 1996.

[21] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. Building linkedin's real-time activity data pipeline. *IEEE Data Eng. Bull.*, 35(2):33–45, 2012.

[22] Bobby Woolf Gregor Hohpe. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional, 2 edition, 10.

[23] Supun Kamburugamuve. Survey of Distributed Stream Processing for Large Stream Sources. https://www.sics.se/~amir/files/download/dic/2013%20-%20Survey%20of%20Distributed%20Stream%20Processing%20for%20Large%20Stream%20Sources.pdf, 2013. last visited on 05.03.2015.

[24] Martin Kleppmann. Stream processing, event sourcing, reactive, cep... and making sense of it all. http://blog.confluent.io/2015/01/29/making-sense-of-stream-processing/. last visited: 11.03.2015.

[25] Jay Kreps. The log: What every software engineer should know about real-time data's unifying abstraction. http://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying. last visited: 18.03.2015.

[26] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14', pages 356–361, New York, NY, USA, 2014. ACM.

[27] S. Marlow. *Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming*. Parallel and Concurrent Programming in Haskell: Techniques for Multicore and Multithreaded Programming. O'Reilly Media, 2013.

[28] N. Marz and J. Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. Manning Publications Company, 2015.

[29] Microsoft Corporation (MSDN). Integration topologies. https://msdn.microsoft.com/en-us/library/ff647958.aspx.

[30] B. O'Sullivan, J. Goerzen, and D.B. Stewart. *Real World Haskell: Code You Can Believe In*. O'Reilly Media, 2008.

[31] Andrew S. Tanenbaum; Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2 edition, 10 2006.

[32] Alvaro Videla and Jason JW Williams. *RabbitMQ in action: distributed messaging for everyone*. Manning, 2012.

[33] Steve Vinoski. Advanced Message Queuing Procotol (AMQP). http://steve.vinoski.net/pdf/IEEE-Advanced_Message_Queuing_Protocol.pdf, 2006. last visited on 04.03.2015.

[34] Charith Wickramarachchi, Srinath Perera, Shammi Jayasinghe, and Sanjiva Weerawarana. Andes: a highly scalable persistent messaging system. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 504–511. IEEE, 2012.

List of Figures

List of Tables