FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2019 Open
Author: Ferdinand Rødne Tvedt	Ferdinand R. Tvedt (signature of author)

Programme coordinator: Leander Jehl

Supervisor(s): Leander Jehl

Title of master's thesis:
Strict ordering guarantees for event-source systems

Credits: 30 ECTS

Keywords:

Distributed Systems • Atomic Multicast • Micro-Services • Apache Kafka • Event-Source System • Topics • Partitions

Number of pages: 73

+ supplemental material/other: 5

Stavanger, June 15, 2019

Title page for Master's Thesis
Faculty of Science and Technology

UNIVERSITY OF STAVANGER

MASTER THESIS

Strict Ordering Guarantees for Event-Source Systems

Author:
Ferdinand R. TVEDT

Supervisor:
Leander JEHL

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in the

Department of Electrical Engineering and Computer Science

June 14, 2019

UNIVERSITY OF STAVANGER

Abstract

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Master of Science

Strict Ordering Guarantees for Event-Source Systems

by Ferdinand R. TVEDT

The digital world is constantly developing globally, more people are connecting to the internet every day and companies that used to be national businesses are developing into international businesses. The constant stream of new users being able to use the internet increases the demand for both old and new service providers. To cope with the potential millions of users companies relies on breaking their services up into micro-services, on multiple different servers providing the users with high availability, low latency, and disaster tolerance(i.e data center failure). Micro-services does provide many benefits for the providers, but it introduces a difficult challenge, how to send messages to multiple replicas in a strict order.

Many service providers such as Facebook, Netflix, and SoundCloud rely on event-source systems such as Apache Kafka, allowing micro-services to subscribe to topics and let them be decoupled from each other. If strict ordering was possible with Kafka many providers would not have to develop complex atomic multicast systems to preserve message ordering.

This thesis describes how it is possible to achieve atomic multicast by using Kafka and demonstrates the capabilities by an implementation we call AtomicKafka. AtomicKafka is a content based strict ordering event-source system that implements an atomic multicast algorithm. We also present an optimized algorithm specifically for Kafka to utilize the way Kafka partitions topics to increase performance. The results of the evaluation show that when sending 40.000 messages we got an baseline average of 83.6 deliveries per second. For the evaluation we were able to reach an average of 26.6 deliveries per second with 10% AMCast messages, and 15.25 deliveries per second for 50% AMCast messages.

Acknowledgements

I would like to thank my family and for all their support and patience during my Master's degree.

I would also like to thank Leander Jehl for supervising the thesis and for his dedication to my work. The weekly meetings and discussions provided me with invaluable feedback and have been very helpful pointing me in the right direction.

Contents

Abstract iii

Acknowledgements v

1 Introduction 1

2 Background 5

2.1 System Model . . . 5
2.2 Atomic Broadcast . . . 5
2.3 Atomic Multicast . . . 7
Non genuine vs genuine atomic multicast . . . 8
2.4 Apache Kafka . . . 9
Topics . . . 10
Partitioner . . . 10
Producer . . . 11
Consumer . . . 11
Messages . . . 12
2.5 Earlier work . . . 12

3 Design and Method 15

3.1 Design . . . 15
System components . . . 15
Messages . . . 16
Data structures . . . 18
3.1.1 Algorithm: Proof of Concept . . . 21
Simple Example . . . 26
Detailed Example . . . 27
3.1.2 Optimization . . . 31
Simple Example . . . 32
3.1.3 Enabling partitions for AtomicKafka . . . 34
Merger . . . 37
3.2 Failure handling discussion . . . 39

3.2.1 Rebuilding the local message state map 39
Rebuilding LSMS: reading from beginning-to-end of topic 40
Rebuilding LSMS: reading from end-to-beginning of topic 40
3.2.2 Client takeover 40

4 Implementation 43
4.1 AtomicKafka 43
4.1.1 System Architecture 43
4.1.2 Consumer 45
4.1.3 Producer 46
Configuration 47
Custom partitioning scheme 47
4.1.4 Messages 47
4.1.5 Message Format 48
4.2 AtomicKafka Client 49

5 Evaluation 51
5.1 Hardware setup 51
5.2 Evaluation setup 52
5.3 Results 53
5.3.1 Baseline 53
5.3.2 90-10 Evaluation 55
5.3.3 80-20 Evaluation 57
5.3.4 70-30 Evaluation 59
5.3.5 60-40 Evaluation 61
5.3.6 50-50 Evaluation 63

6 Discussion and further work 67
6.1 Discussion 67
6.1.1 Performance 68
6.1.2 Further work 70

7 Conclusion 73

A Program Code 75

B Kafka Broker Configuration 77

Bibliography 81

List of Figures

2.1 Atomic Broadcast Properties . . . 6
2.2 Atomic Multicast Properties . . . 7
2.3 Atomic Multicast phases.[16] . . . 7
2.4 AtomicKafka architecture . . . 9
2.5 Topic partitions . . . 10

3.1 AtomicKafka interaction . . . 16
3.2 Priority-Queue with conflicting timestamp . . . 21
3.3 First design message propagation . . . 27
3.4 Example of phases and messages . . . 30
3.5 AMCast optimized message propagation . . . 31
3.6 Topic Layering . . . 35
3.7 Partitioned AtomicKafka . . . 36

4.1 AtomicKafka Architecture . . . 44
4.2 AtomicKafka Client Architecture . . . 49

5.1 Baseline: Incoming messages per second . . . 53
5.2 Baseline: Outgoing messages per second . . . 54
5.3 Baseline: Deliveries per second . . . 54
5.4 10% AMCast: Incoming messages per second . . . 55
5.5 10% AMCast: Outgoing per second . . . 56
5.6 10% AMCast: Deliveries per second . . . 56
5.7 20% AMCast: Incoming messages per second . . . 57
5.8 20% AMCast: Outgoing messages per second . . . 58
5.9 20% AMCast: Deliveries per second . . . 58
5.10 30% AMCast: Incoming messages per second . . . 59
5.11 30% AMCast: Outgoing messages per second . . . 60
5.12 30% AMCast: Deliveries per second . . . 60
5.13 40% AMCast: Incoming messages per second . . . 61
5.14 40% AMCast: Outgoing messages per second . . . 62
5.15 40% AMCast: Deliveries per second . . . 62
5.16 50% AMCast: Incoming messages per second . . . 63

5.17 50% AMCast: Outgoing messages per second 64
5.18 50% AMCast: Deliveries per second 64

Listings

3.1 Message Definition 18
3.2 Message composition 20
3.3 LMSM composition 29
3.4 Message composition for partitions 37
4.1 Consumer Configuration 46
4.2 Producer Configuration 47
4.3 AtomicKafka Client message example 48
4.4 Control message example 49

List of Abbreviations

PoC	**P**roof **O**f **C**oncept
AMCast	**A**tomic **M**ulticast
Msg	**M**essage
Msg/s	**M**essages **P**er **S**econd
API	**A**pplication **P**rogramming **I**nterface
LSMS	**l**ocal **M**essage **S**tate **M**ap

List of Symbols

Π	**A**tomic**K**afka **S**ystem
Ω	**K**afka **S**ystem
p_i	**S**ingle **P**rocess in Π
$\mid p \mid$	**N**umber **o**f **P**rocesses
T	**C**ollection **o**f **K**afka **t**opic
tp_i	**T**opic **P**artition
$\mid tp \mid$	**N**umber **o**f **T**otal **P**artitions
$\mid topic \mid$	**N**umber **o**f **K**afka **t**opics

Chapter 1

Introduction

The digital world is becoming more globalized by every day that goes by, and the demand for distributed systems grows as more services on the market creates more data that needs to be produced, stored and consumed by users. In less than two decades there has been a significant increase in services relying on storing and serving vast amounts of data, services such as SoundCloud, Uber, Netflix, Amazon, eBay and other similar providers. While all of them is providing users with a different type of content or services, they offer their content or services on the internet for users. One of the challenges creating such systems is to remain responsive, available and keeping the data consistent regardless of how many users is generating a load on their services.

To overcome this challenge and handle thousands if not hundreds of thousands of users simultaneously they depend on micro-service architecture[7, 30, 31, 37] and not monolithic architecture for their systems. The micro-service architecture allows SoundCloud, Uber, Netflix, Amazon and eBay to develop loosely coupled, independently deployable applications[35] that can focus on one task, instead of having a complex monolithic application that has to be able to handle everything.

By using micro-services Netflix and the other providers can handle all the users simultaneously by scaling their micro-services horizontally[5], distributing their micro-services across multiple servers increasing the reliability and availability of their systems. The challenge by distributing their micro-services across multiple servers is ensuring that the data is strongly consistent across the servers in the case of server failure or ongoing maintenance. Distributed systems today rely on micro-services to hide the fact that some micro-services fails, distribute incoming loads over all the micro-services and keeping response times low for users. Though micro-services can increase reliability and availability, it makes keeping strong consistency

across micro-services a challenging problem, which is a problem that has been a challenge among researches at least since 1989[34].

When Amazon implemented Dynamo their goal was to create a distributed key-value storage system with an "always-on" experience, making it highly available for their users. To achieve this kind of availability Amazon decided to weaken the consistency guarantee of the system[10]. By weakening the consistency guarantee the services are placing the responsibility of coping with inconsistent service behavior, making the application decide on the correctness of the data received from the services.

Netflix has also stated that they wish to use Kafka to collect test results and be able to replay the events similar to a state machine[8], but they do not say anything about how they intend to achieve this. We this as an excellent reason to research the possibility to implement atomic multicast in Kafka and to see how an atomic multicast enabled Kafka performs.

There are multiple proposals[9, 15, 18, 28, 25] and implementations[29] that solves the strong consistency challenges by using atomic multicast (AM-Cast). Some of the proposals[18, 28] are static, meaning that if one needs to add or remove a multicast group one would have to shut down the entire distributed system before the reconfiguration. By shutting down the system we break the user's illusion of "always-on". It is also tough to implement these proposals as they are very complex and it is easy to make mistakes. Benz et al. proposed a dynamic atomic multicast[4] where groups can subscribe and unsubscribe to data streams without having to shut down for reconfiguration. The downside of this proposal is that the groups are tightly coupled, meaning that messages being sent to a stream will always reach the group members, even though the data might not always be of use for a process in the group. What we want to find out is if it is possible to develop an atomic multicast system using preexisting systems that the industry is currently using, such as event source system as Kafka. Moreover, if it is possible what is the cost of introducing atomic multicast to such a system.

To overcome these challenges we introduce AtomicKafka, a system that uses Kafka as a foundation to achieve genuine atomic multicast for the content-based event-source system. Kafka is a popular event-source system used by many providers such as Microsoft, Airbnb and Netflix[36], where Kafka handles approximately around 700 billion messages per day for Netflix[6]. To be able to handle this amount of data Netflix are using approximately around 150 clusters consisting of 3500 instances.

The current version of Kafka(2.2) preserves the order of messages within a single partition as long as the producer sends them in a specific order, meaning that if multiple partitions/topics exist it is not able to keep the strict ordering guarantee[24]. So our challenge was to find out if it was possible to achieve AMCast by using Kafka, and what the performance cost of introducing AMCast on Kafka. Our intention with AtomicKafka is that it should become a system that is more flexible than Kafka and the proposed atomic multicast systems[9, 15, 18, 28]. We want AtomicKafka to be able to have both strict and weak consistency depending on the need of the users and dynamically adding and removing of multicast groups without downtime. We also want it to make it easier for developers as they do not have to develop complex systems by a widely used event-source system which removes much of the complexity.

As previously mentioned some of the proposed systems[18, 28] cannot reconfigure their multicast groups without having to shut down the systems. The other proposals[4, 25] can reconfigure the multicast groups without having to shut down the system, but the multicast groups themselves are not dynamic, making them tightly coupled. A tightly coupled multicast group has the disadvantage that if a message needs to reach two services which are not apart of a group by themselves the message would have to be atomic multicast to a service which does not involve the message. This behavior of tightly coupled multicast groups resolves in partially genuine atomic multicast system[25]. For AtomicKafka there are no predefined multicast groups, in AtomicKafka a multicast group is created dynamically based on the messages intended receivers. By doing this AtomicKafka can AMCast a message regardless of intended receivers, meaning that atomic multicast messages are only handled by processes they are intended for. Since the messages decide which topics are the receivers AtomicKafka can dynamically add or remove topics in runtime, allowing AtomicKafka to be smoothly integrated with Kafka.

Our contribution is an implementation of AtomicKafka based on the Algorithm 3. We were able to evaluate the system and estimate the cost of performance when introducing AMCast to Kafka. With a baseline of 83.6 deliveries per second we were able to have an average of 26.6 deliveries per second with 10% of the messages being atomic multicast messages. When we increased the number of atomic multicast messages to 50% we achieved a delivery rate of 15.25 per second.

Chapter 2

Background

This chapter will give the reader an introduction to the theory, terminology, and technology used to solve the problems explained in chapter 1.

2.1 System Model

Normally a distributed system[4, 9, 15, 18] Π is considered to consist of interconnected processes $\Pi = p_1, p_2 \dots p_n$ which communicates through broadcasting messages between each other through a reliable broadcast. In this thesis we assume that the distributed system Π consist of decoupled processes $\Pi = p_1, p_2 \dots p_n$ where they communicate using a intermediate system Ω to communicate with each other. We assume that Ω does not exhibit any failures.

A process may experience a failure, but do not exhibit any Byzantine failures. If a process fails it is because the process has crashed and could recover from this.

Processes are considered either correct or faulty. A correct process could eventually be operational forever, and can reliably send and receive messages from Ω as long it is correct, which means that the process is correct for long enough to terminate an instance of consensus.

2.2 Atomic Broadcast

In distributed system broadcast communication is a fundamental problem that is reaching consensus among multiple different processes[11].

Abstractions are used to propagate messages to different processes. There exist many different communication abstractions that can guarantee different properties for the abstractions such as varying degree of consistency, reliability and availability. Abstractions such as Regular Reliable Broadcast, FIFO

Broadcast and Causal Broadcast, which are all similar to each other but has different applications and tries to solve different problems.

Atomic Broadcast is another broadcast abstraction that is similar to both FIFO and causal Broadcast as all three enforce a specific delivery behavior based on the order of messages. Where FIFO ensures when a process π_1 broadcasts messages they will be delivered in the same order as they were broadcasted, but only for p_1, if p_2 also broadcasts messages it does not ensures an order between the two processes.

Causal Broadcast works similarly as it ensures a global ordering for all messages that causally depends on each other. Meaning that if m_2 has been delivered and both m_1 and m_2 has a dependency on each other then m_1 has been delivered before m_2. The issue here is that if two messages are unrelated it will not ensure a global order on the unrelated messages.

Atomic broadcast unlike Causal and FIFO broadcast enforces a global order regardless of which processes sent the messages and the causality of the messages. Atomic broadcast ensures that every process $\Pi = \{p_1, ..., p_n\}$ in the system will deliver the same messages $\{m_1, m_2, ..., m_n\}$ in an global order, meaning that every process will have the exact same ordering of the messages $\{m_1, m_2, ..., m_n\}$ locally.

- Validity: If a correct process p broadcasts a message m, then p eventually delivers m.
- No duplication: No message is delivered more than once.
- No creation: If a process delivers a message m with sender s, then m was previously broadcast by process s.
- Agreement: If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
- Total order: Let m1 and m2 be any two messages and suppose p and q are any two correct processes that deliver m1 and m2. If p delivers m1 before m2, then q delivers m1 before m2.algorithm we will be using.

FIGURE 2.1: Atomic Broadcast Properties

2.3 Atomic Multicast

Atomic Multicast is a communication abstraction very similar to Atomic Broadcast in that both abstractions ensure global sequence ordering on messages. The most significant difference between the multicast version and broadcast version is where Atomic Broadcast will propagate the message to all processes in the system Π atomic multicast can select a group of processes $G_i \subset \{p_1, p_2\}$ to propagate the message instead of all processes.

To achieve Atomic Multicast a system has to adhere to the Atomic Multicast properties 2.2 defined below.

- Validity: if a correct process p multicast a message m, then eventually all correct processes $g \subset m.dst$ deliver m.
- Agreement: if a correct process p delivers a message m, then eventually all processes $q \subset g$, deliver m.
- Integrity: For any process p and any message m, p delivers m at most once(no duplicates), and only if $p \subset g$, $g \subset m.dist$, and m was previously proposed.
- total order: For any two message m and m′ and any two processes p and q such that $p \subset g$, $q \subset h$ and $g, h \subseteq m.dst \cup m'.dst$, if p delivers m and q delivers m′, then either p delivers m′ before m or q delivers m before m′.

FIGURE 2.2: Atomic Multicast Properties

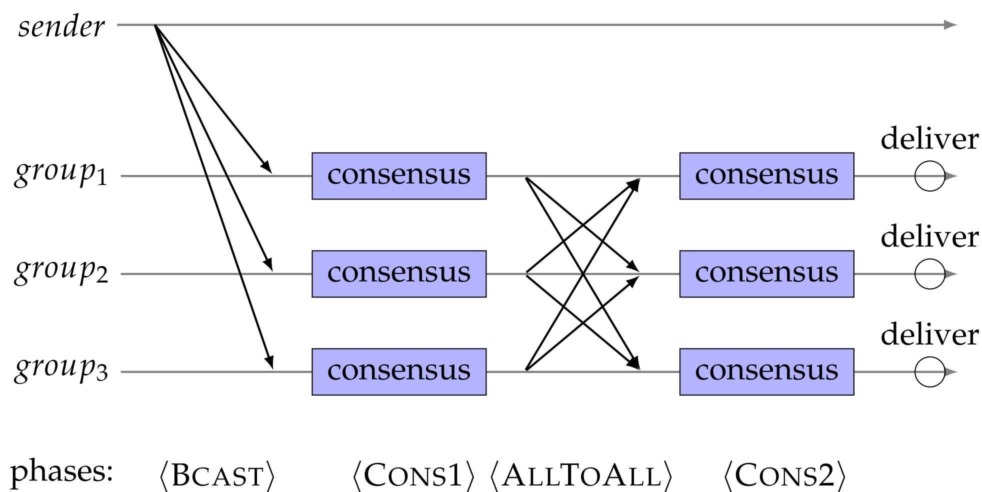

FIGURE 2.3: Atomic Multicast phases.[16]

The figure 2.3 above taken from the article 'Decoupling atomic multicast'[16] shows the different atomic multicast phases. Typically it requires four phases where the first message is sent to every receiving group. When this message is received each group will add a timestamp on the message and agree on a timestamp. Then each group will inform every other group about their decision in the All to All phase. In the last consensus round they all will agree to the largest timestamp among the groups and all groups end up delivering the same message order.

When comparing Atomic Multicast to the Atomic Broadcast they are very similar, though Atomic Multicast has two properties that are defined in a more detailed way. Both the integrity and total order properties are given more details regarding groups of processes receiving a message instead of the whole system. Comparing the Atomic Multicast Properties 2.2 and Atomic Broadcast Properties 2.1

Non genuine vs genuine atomic multicast

As explained in Chapter 2.2 atomic broadcast enables a message to be sent to all processes in a system along with the guarantees that all correct processes agree on a specific sequence on the messages they deliver.

The opposite of atomic broadcast is atomic multicast, which can target a subset of processes which agrees on a specific sequence on the messages they deliver.

This is why it is possible to use atomic multicast to implement atomic broadcast, by only atomic multicast messages to all processes[15]. It is also possible to achieve atomic multicast by implementing atomic broadcast if the processes drop messages not intended for themselves.

Though by using atomic broadcast to implement atomic multicast would violate the minimality property. The minimality property states that only the sender and the addressed receivers of a message should be involved in the protocol that is needed to be able to deliver the message[15]. It is evident that an atomic multicast algorithm using atomic broadcast does not satisfy this property, as every process would have to process the message even those who are not intended to process it.

2.4 Apache Kafka

Kafka is a distributed event-source system, which we are using to enable us to simplify our process to implement the Atomic Multicast algorithm we will be using. Kafka is a distributed, replicated, log service created by LinkedIn and open-sourced in 2011 [27]. Kafka was created to enable tracking of events created from LinkedIn websites such as page-views, keywords, and ad views so that LinkedIn could monitor their user's engagement towards the site. Kafka differs from other Publish-Subscribe system as Kafka is not an implementation of the MQTT or AMQ protocols. LinkedIn created its own Kafka protocol[12] that is designed explicitly for Kafka usage.

Since we will be using Kafka as our foundation an explanation how the system works is necessary to get the fundamental understanding of how our algorithm is going to work and how by using Kafka we can ensure many of the properties of Atomic Multicast without having to develop a complex solution of our own, and since Kafka is widely used in the industry it might be helpful for other developers to understand how they could achieve Atomic Multicast using either Kafka or their already established systems.

Kafka is usually run as a distributed cluster, illustrated by Figure 2.4 where each node is called a broker, illustrated by the white rectangles in Figure 2.4. Brokers are nodes which can contain multiple topics, illustrated by red, green, and blue rectangles, where each topic can act as a leader for the topic which provides fault tolerance for topics and high availability. A Kafka producer is illustrated by an orange circle in Figure 2.4. The producer is responsible for creating and sending messages into a topic or multiple topics residing inside the Kafka cluster so that Kafka consumer can fetch those massages. The red circles represent the Kafka consumers. A consumer in Kafka is responsible for consuming Kafka messages from a topic it is subscribed too, explained in Chapter 2.4.

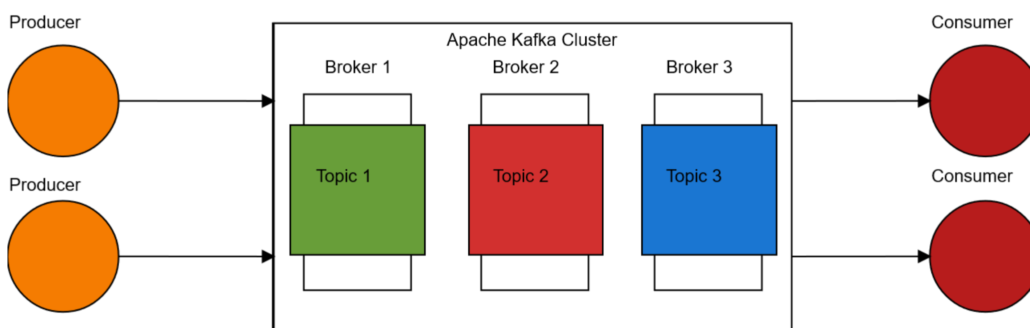

FIGURE 2.4: AtomicKafka architecture

Topics

In an event-source system like Kafka messages will always have a designated address called Topics. In Kafka a topic is defined as $T = \{tp_1, tp_2, ..., tp_n\}$, meaning there can exist multiple partitions [24]. This is visualized in the Figure 2.5, where it shows that a client produces a message to Kafka, it is assigned a partition in Kafka and eventually it is consumed by a client again. The functionality of how it is partitioned is explained below in Chapter 2.4.

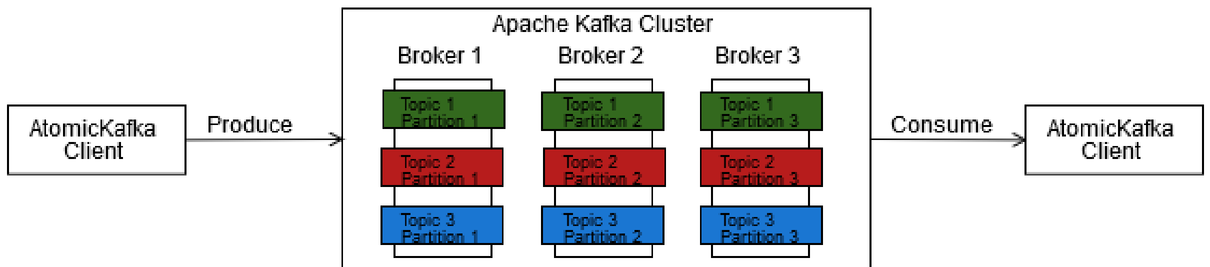

FIGURE 2.5: Topic partitions

When a producer is sending a message to Kafka it will have what is called a topic leader, a partition with the designated responsibility of coordinating new messages, if partitioning is enabled. When the leader has accepted the new message it will update any existing replicas with the new message.

A replica is a backup of a topic residing in another broker, with the new data, and if the leader fails one of the replicas will take over the responsibility and become the new leader for that topic.

In a Publish-Subscriber system, a collection of messages is sorted into designated locations called topics. Both the publishers and subscribers choose which topics they want to either send or receive messages from. Since Kafka is a distributed system, the topics can also be distributed by using either partition, replications, or both.

"*More Partitions Lead to Higher Throughput*.[26]"

Partitions are mainly used to increase the throughput of Kafka system as it allows the Kafka brokers to handle multiple producers producing messages to multiple partitions, instead of a single topic.

The replications is a copy of the topic or partition residing on another broker in case of broker failure.

Partitioner

The partitioner is a schema that is provided to a Kafka producer, defining how messages should be partitioned. As explained above in Chapter 2.4,

partitions are essential if one wants to increase performance. In Kafka the default partitioner has two ways to decide which partition a message belongs too.

If a message is assigned a key the partitioner will calculate a hash value of that key and use that to decide the partition it belongs too. If no key is defined it will then assign the partitions in a round-robin fashion.

It is also possible to create a custom partitioner as we explain in Chapter 4.1.3.

Producer

A client that produces a message that is sent into a Kafka cluster is known as a producer. A producer can be thought of as a data source which subscribes to a topic it will send messages to.

The producer can send messages to the broker which is the leader of the topics directly.

The producer can directly communicate with the broker since it is allowed to send a request for information about which brokers is alive and which broker is the leader for the desired topic to any of the Kafka nodes at any time [13].

The producer also includes support for idempotent[13] delivery. Idempotent delivery means that the messages sent using this option is delivered exactly once to a single topic during the producers lifetime. Producers also support transactional[13] delivery which allows the user to send messages to multiple topics such that all the messages are delivered or none of them are, which is an atomic operation.

Together these two capabilities enable the producer to use the exactly once semantics[13] for Kafka.

Consumer

A client that consumes messages from a Kafka cluster is known as a Consumer. If a consumer wants to consume messages from certain topics it has to notify the brokers about subscribing to the single or multiple topics it wants to consume from. Each consumer when subscribing to a topic will notify the broker about an identity called group identification. This ID enables the brokers to keep track of which messages it retains has been received and read from the specific consumer, and in the case of a disconnect the consumer is

not needed to reread every message and instead start reading from where it disconnected.

Messages

The messages used by Kafka is simple in design. A Kafka message is composed of a Key and a value where both can be of any object as long as a serialization method is defined. If a key is defined for the value the key will be used by the partitioner, explained in Chapter 2.4, deciding the partition it belongs to.

2.5 Earlier work

Before we started working on this thesis we conducted an project[32] to familiarize ourselves with Kafka and atomic multicast. The architecture of the framework from that project is the one used as the basis in this thesis. The architecture is explained in details in Chapter 4.1.1.

We attempted to implement an atomic multicast algorithm in that project and concluded it was possible, though while working on this thesis we found that the conclusions from that project needed to be discarded. The atomic multicast implementation in that project was faulty, and we had assumed that the timestamp generated by Kafka was synchronized timestamp shared between the brokers. That assumption was wrong[13], and while working on this thesis we spent much of our efforts solving this issue, further details in Chapter 6.1.1. The implementation in the project ended up working because of the servers internal clock was synchronized at the time of evaluation, and when we started working on this thesis the internal clocks were no longer synchronized.

In this thesis we have rebuilt the entire consumer implementation to work with logical clocks, optimized the message ordering by using a binary heap, making delivery checking easier as we do not have to iterate over every message in the local state map, explained in Chapter 3.1, for ordering deliverable messages. We have also moved away from the inefficient transactional producer, that promises atomicity operation, ACID atomic and not Lamport Atomic when sending messages to multiple topics. We also found that the algorithm in itself had issues while using multiple producers, to compensate for this the Algorithm 3.1.1 was redesigned to function with multiple producers.

Because of these issues, we discard the results of that project as invalid, though we gained useful insights in Kafka, the Kafka API and atomic multicast theory.

Chapter 3

Design and Method

3.1 Design

This chapter introduces three different designs to enable atomic multicast for Kafka. We present a Proof of Concept(PoC) design which enables atomic multicast for any event-source System. After reviewing the PoC we investigate the potential optimization of the PoC algorithm resulting in the second design. The third design is a Kafka specific algorithm that adheres more to Kafka design principles than the previous two designs. Lastly, we explain how to recover from a node failure by discussing multiple strategies such as rebuilding the state of the node.

System components

As mentioned in Chapter 2 Kafka is an open-source project, and the code is hosted on GitHub[2]. That means we could have forked the project and used that for our implementation of AMCast. That approach has two significant disadvantages, maintainability and complexity. To be able to maintain the AMCast feature one would have to keep merging newly implemented features from Kafka into AtomicKafka. The chance of breaking AMCast functionality is more prominent, making maintainability harder. We would also have to make sure that we do not break any other functionality of Kafka, making it more complex to implement. The other hindrance for this approach is the size of the Kafka project. At the time of writing the current commit in the repository consist of 384 669[1] lines of code, making it impractical to implement AMCast directly into Kafka.

Our solution is to design a system that is decoupled and interacts with Kafka both as a consumer and a producer client. In Figure 3.1 we provide an overview of what the system looks like from a top-level view. The two

orange circles are regular Kafka producers and the two red circles are regular Kafka consumers, as explained in Chapter2.

The AtomicKafka cluster in Figure 3.1 has three nodes which are subscribed to topics 1, 2, and 3. This means that any AMCast message sent to either of those topics will use atomic multicast, if not it will be ignored.

There are three more topics marked yellow, and these are topics where atomic multicast is never needed but was included to show that it is possible to have regular topics too.

The AtomicKafka client is very similar to a regular Kafka consumer, the issues of using a regular Kafka consumer on topics that AtomicKafka nodes are subscribed too is the amount of control messages are consumed. The AtomicKafka client can filter out all the control messages and only give the end user the delivered messages instead.

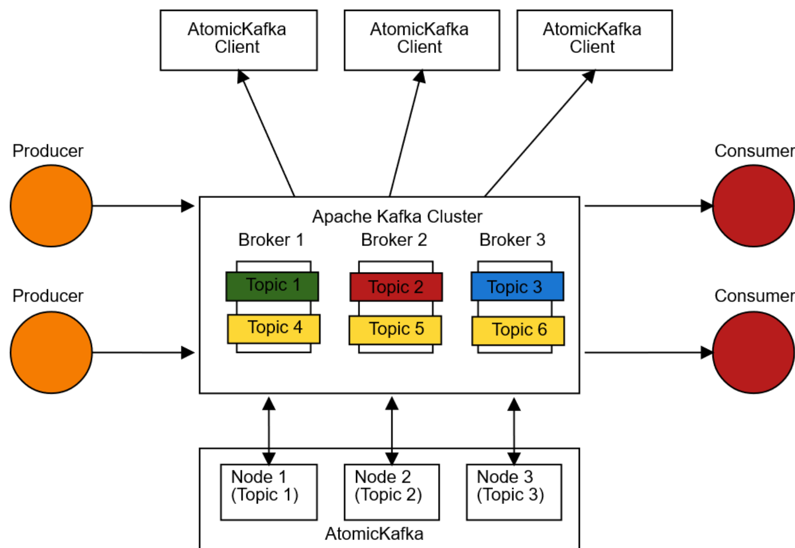

FIGURE 3.1: AtomicKafka interaction

For the design we assume there will exist one node per Kafka topic that requires AMCast functionality. Meaning there will exists $| p_i | <= | topic_i |$ number of AtomicKafka nodes. To simplify we will assume from here on that our system requires that all topics require AMCast, meaning we will need $| p_i | == | topic_i |$ number of nodes.

Messages

We used a specific message format when using AMCast with AtomicKAfka, that includes both message content and metadata, as can be seen in Listing 3.1. Many fields have to be populated in a message, but a client only needs

to populate messageID, senderID, value, topic and type. Though a client is required only to send a message of type ClientMessage. There is a possibility of misusing this by sending malformed client messages, but in our current design, we assume that every client is honoring this requirement and validating this is left to the implementation. All other message types are used by AtomicKafka internally. Those messages are used to determine the message state and how AtomicKafka should proceed to process the message. To do this the system implements the five different message types as listed below in 3.1.

LISTING 3.1: Message Definition

MessageID
The ID is used as a unique identifier during the AMCast.

Timestamp
A Lamport clock used to decide in what order the message was received.

SenderID
The sender ID is mostly used as a identifying field for debugging purposes.

Value
The data or message intended to be sent.

Topic
The recipients of the message. This field can contain one or more recipients.

MessageType
There are multiple message types used by AtomicKafka:

- **Client Message**
  This is a message sent from an AtomicKafka client to Kafka topic, and is the only type AtomicKafka client is able to send.
- **Notify Message**
  This is a control message used to forward a single message to other nodes that are marked as recipients.
- **Acknowledge Message**
  This is a control message used to acknowledge that a node has received an notify message.
- **Decided Message**
  This is a control message used in the optimized algorithm. This is a message sent from the node that decides on the lamport timestamp.
- **Delivery Message**
  This is the final message that a client can consume, this message will be in total order across the topics marked as recipients.

Because AtomicKafka uses control messages like a regular consumer would consume every message possible from a topic, because of this we have designed a custom client called AtomicKafka Client, as seen in Figure 3.1. This client is capable of sending ClientMessages into Kafka and read the delivered messages, filtering out the control messages.

Data structures

To enable AMCast for AtomicKafka we need a mechanism to ensure the correct sequence of deliveries. To keep track on the sequence messages should

be delivered and in what order we create hashmap, that we call a local message state map (LMSM). This LMSM uses the messageID as a key and for the value it creates a new hashmap so $< key, hashmap >$ pair. The hashmap stored in the value is a collection of every message that is related to messageID. This hashmap is using the senderID as key and the message itself as a value, $< int, String >$. LMSM maps all messages with the same messageID into a new map, so it has a message state for every node with the messageID.

For example, if we have a message formatted as

$$< senderID, messageID, timestamp, [topics], messageType, value >$$

and a topic $topic_1$ receive the message

$$< sender_1, messageID_1, timestamp_2, [topic_1, topic_2], ClientMessage, "hello" >$$

p_1 will create a new entry on key 1. This results in creating a new hashmap $< 1, < int, message >>$, and store the nodes own message state as

$$\begin{aligned}
&< messageID_1 \ , \\
&\quad [\ < senderID_1, \\
&\quad\quad (senderID_1, messageID_1, timestamp_2, [topic_1, topic_2], AckMessage, "hello") > \\
&\quad]>
\end{aligned}$$

p_1 will also create a notify message and send it to $topic_2$ where p_2 process the notify message in the same manner as p_1 handled the client message. After a local state has been created for p_2 it will reply with the message

$$< sender_2, messageID_1, timestamp_5, [topic_1, topic_2], AckMessage, "hello" >$$

resulting in p_1 updating the LMSM to

$$\begin{aligned}
&< messageID_1 \ , \\
&\quad [\ < senderID_1, \\
&\quad\quad (senderID_1, messageID_1, timestamp_1, [topic_1, topic_2], AckMessage, "hello") > , \\
&\quad\ < senderID_2, \\
&\quad\quad (senderID_2, messageID_1, timestamp_2, [topic_1, topic_2], AckMessage, "hello") > , \\
&\quad]>
\end{aligned}$$

By using the LSMS a node can check if it has received all the acknowledgments needed to decide when a message is deliverable. Choosing a timestamp is done by iterating over all messages under messageID, and the timestamp with the largest value is chosen. To do this a timestamp is created from a logical clock every time a topic receives a ClientMessage or NotifyMessage. The Lamport clock of a node can be updated choosing the highest value between the message timestamp and local clock, otherwise the Lamport clock is increased linearly for Client-, Notify-. and AckMessages, creating a local ordering of when a message was received.

As the LSMS only keeps track of message state a priority-queue is used to keep track of the order in which a message is to be delivered. The priority-queue contains each messageID only once, so we have a reference to the message location in LSMS. As messages are received by a $node_i$ the message is added into the priority-queue using the timestamp as the key and the message object reference as value. To keep the messages in order the priority queue automatically orders the messages based on their timestamp, the lower the timestamp value is the higher priority the message gets in the queue. In the case of two different messages has the same timestamp a tiebreaker is made on the messageID, where the lowest messageID value has a higher priority.

LISTING 3.2: Message composition

$m_1 := < senderID_2, messageid_1, timestamp_1, [topic_1, topic_2], ACKNOWLEDGE, "one" >$

$m_2 := < senderID_2, messageid_2, timestamp_1, [topic_1, topic_2, topic_3], ACKNOWLEDGE, "two" >$

$m_3 := < senderID_2, messageid_3, timestamp_4, [topic_1, topic_2, topic_3], ACKNOWLEDGE, "three" >$

$m_4 := < senderID_2, messageid_4, timestamp_9, [topic_1, topic_2, topic_3], ACKNOWLEDGE, "four" >$

If a process p_2 is pushing the message above into a priority-queue it will end up having the structure as the Figure 3.2.

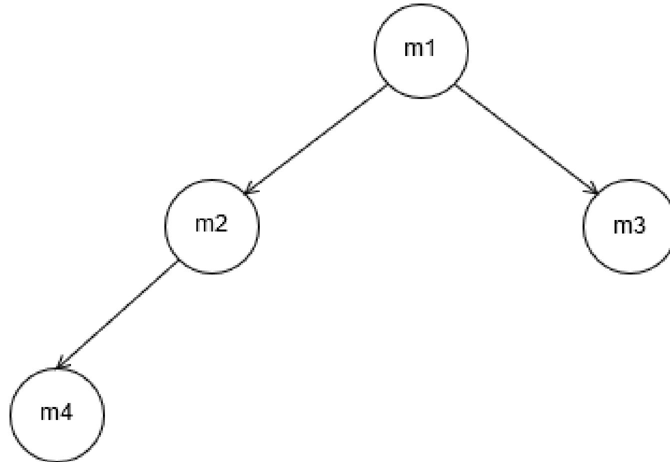

FIGURE 3.2: Priority-Queue with conflicting timestamp

As messages have been processed, a method should execute to check if the first message in the priority queue is annotated as Delivery. If every node has acknowledged the message a decision can be made and the message will then be deliverable. As the message is deliverable and the message position in the queue is at the top every $node_i$ will pop the the message from their priority-queue and send it to the topic $node_i$ is subscribed to. After the message has been delivered to $topic_i$ each $node_i$ will then remove all the messages placed in the messageID key from the LMSM.

Looking at Figure 3.2 if each message was annotated as deliverable the queue send the messages in the order m_1, m_2, m_3, m_4 to the topic p_i is subscribed too.

3.1.1 Algorithm: Proof of Concept

The PoC design is similar and behaves just like other AMCast algorithms[18, 9, 16] designed from before. However, since we are creating a system that relies on Kafka as an Event-Source System, much of the complexity regarding atomic multicast is handled. Specifically validity, no creation and agreement [18, 9, 16] is already taken care of. Allowing us to focus on the AMCast algorithm.

The first design of AtomicKafka is designed as a PoC without considering any optimizations. The design is to ensure that a decoupled event-source system such as Kafka is capable of using AMCast and of evaluating the system design.

The Algorithm 1 outlines how we attain AMCast for an Event-Source System, in our case Kafka. Two figures also accompany this algorithm. Figure 3.3 shows the different phases that a message has to go through before it can

be delivered. Figure 3.4 shows an example on how AMCast works step by step.

When an AtomicKafka process is starting up it will have a reference to both a normal Kafka consumer and Kafka producer service as seen from Line 1. The first method to run in Algorithm 1 is the Initialization on Line 5. The first variable to be initialized is the map variable. The map variable is the LMSM that consists of a Hashmap with the data-structure detailed in Chapter 3.1. The ID is used to uniquely identify different AtomicKafka nodes when a message is sent from an AtomicKafka node the ID is added to the sender field. The ID has to be globally unique, as it is also used as a key for retrieving messages from the LMSM.

The priority-queue is a Binary heap structure also known as a priority-queue. The priority queue is used to arrange a message order as explained in 3.1.

The timestamp is a Lamport Clock[17] also known as a logical clock. We use this timestamp to register when a message has arrived in an AtomicKafka node.

When the node has finished initializing it will wait for a message to be received by the topic it is subscribed too. The method on Line 11 handles all the messages that are being sent from a client to AtomicKafka. As a message is handled by the Received-ClientMessage method it is given a timestamp and the timestamp counter is increased on Line 12 and 13. The message is added to the LMSM on Line 14. If this is the first time the node has seen this message before it will create a new hashmap creating the data structure that was explained above in Chapter 3.1. Once the message state has been set a reference to the message is also added to the priority-queue on Line 15. After the message has been assigned a timestamp and added to the hashmap and binary heap it is inspected to see if the node is the only receiver or if there are multiple receivers on Line 17. If the node is the only receiver it does not have made an agreement with the other nodes to deliver the message and is marked as type delivery on Line 17, and the reference is updated in the priority queue on Line 18.

If the message has multiple receivers the node will iterate over each topic listed as a receiver on Line 20 to notify other nodes about the message. Line 21 checks if the current topic in the iteration is a topic the node is subscribed too. If the test is true it updates the message type on Line 22 and update message state map on Line 23. Since this message is intended for the node itself it does not need to waste bandwidth or Kafka processing power to send

the message. If the test on Line 21 is false it will update the message type to a Notify message on Line 26. Line 28 triggers the Kafka producer to send the message to the current topic in the iteration.

The method on Line 28 is responsible for handling every Notify messages sent from other nodes. As this is the very first time the node has seen the message it will process it similarly as Line 29 and 30 adds a timestamp to the message. Line 32 puts the message into the LMSM, of an entry does not already exist the node creates a new entry for the message.

Line 32 check if the node is the only receive. As in the Received-ClientMessage method on Line 11 there is a check in place to see if the node is the only receiver of the message, reducing the number of messages sent. The unintended effect of having checks on Line 32 is that it allows the nodes to relay messages between topics, increasing the flexibility of the system. Relaying messages allows users to send messages to arbitrary topics and the AtomicKafka node subscribed to those topics will relay the messages to intended topics.

Line 33 and 34 updates both message type and adds the message to the delivery queue.

If the message contains multiple receivers it will skip Line 32 and continue from Line 36. The message type is updated to Acknowledge on Line 31 to let other nodes know it has received the message. Line 37 adds the message to the queue and Line 38 iterates all the receiving topics. Line 40 triggers the Kafka producer to send out the message to the current topic in the iteration.

The method on Line 5 is invoked when every intended receiver of a message has replied with an Acknowledge to each other. It starts with having an empty message variable on Line 6. The empty variable is intended to hold a reference to a message that will be decided as the delivery message. Line 7 iterates through every Acknowledge message received, using the messageID to retrieve the messages. While iterating through every acknowledge message Line 8 checks if a message has a higher timestamp than the previous message. If this is true the message will be stored in the empty message variable on Line 6. When the message with the biggest timestamp has been found the message type is updated to Delivery on Line 7, and the message inside the delivery queue is also updated. At this point the delivery queue will update the ordering of the stored messages in the same manner as explained in Chapter 3.1 with Figure 3.2.

When a message is marked as Delivery the method in Algorithm 2 is invoked to check if there is any message that can be delivered to a topic or not.

The while loop on Line 16 conditions will prevent the loop from delivering any messages that do not meet the right condition. To be able to loop the first element in the queue has to be a deliverable message if this is true the message is pulled from the queue on Line 17 into a temporary variable. Line 18 triggers the producer to send the message to the topic the nodes is subscribed to. Since the message now has been delivered there is no more need to keep track of the message and Line 19 and 20 removes all the messages that belong to the messageID that was just delivered.

Algorithm 1 AtomicKafka: Proof of Concept part I

```
1: Uses:
2: Kafka-Producer instance p
3: Kafka-Consumer instance c
4:
5: on ⟨INIT⟩ do
6:     map := Π                                   ▷ Init message soft state map
7:     id := ℝ                                    ▷ The unique ID of the node
8:     priority-queue := Π ▷ Binary Heap used to sort deliverable messages
9:     timestamp := 0                             ▷ Lamport clock
10:
11: on event ⟨RECEIVED-CLIENTMESSAGE, msg⟩ do
12:     msg.timestamp = timestamp                 ▷ Add Lamport clock to message
13:     timestamp = timestamp +1                  ▷ Increase Lamport clock
14:     map = map ∪ msg
15:     priority-queue = priority-queue ∪ msg
16:     if Σm.topics == 1∧ m.topics.contains(self.topic) then
17:         msg.type = Delivery
18:         priority-queue = priority-queue ∪ msg ▷ Message is deliverable
19:     else                                      ▷ Find designated topics
20:         for topic ← msg.Topics do
21:             if topic == self.Topic then
22:                 msg.type = Acknowledge
23:                 priority-queue = priority-queue ∪ msg
24:             else
25:                 msg.type := Notify
26:                 Trigger <p, topic, Notify, msg>   ▷ Send to topics
27:
28: on event ⟨RECEIVED-NOTIFY, msg⟩ do
29:     msg.timestamp = timestamp
30:     timestamp = timestamp + 1
31:     map = map ∪ msg
32:     if Σm.topics == 1∧ m.topics.contains(self.topic) then
33:         msg.type = Delivery
34:         priority-queue = priority-queue ∪ msg ▷ Message is deliverable
35:     else
36:         msg.type = Acknowledge
37:         priority-queue = priority-queue ∪ msg
38:         for topic ← msg.Topics do
39:             if topic != self.Topic then
40:                 Trigger <p, topic, Acknowledge, msg>   ▷ Send ACKs
41:
```

Algorithm 2 AtomicKafka: Proof of Concept part II

```
1: Uses:
2: Kafka-Producer instance p
3: Kafka-Consumer instance c
4:
5: on event ⟨RECEIVED-ACKNOWLEDGE, msg⟩ All acks received for messageID do
6:     latestmsg := newMessage()
7:     for storedMsg ← map[msg.id] do
8:         if storedMsg.ts > latestmsg.ts then                ▷ Find biggest ts
9:             latestmsg = storedMsg
10:    latestmsg.type = Delivery
11:    priority-queue = priority-queue ∪ latestmsg
12:    if timestamp < latestmsg.ts then
13:        timestamp = lastestmsg.ts
14:
15: on event ⟨DELIVERY, msg⟩ deliver-heap contains delivery do
16:    While(priority-queue.peak() == Type.Delivery) then
17:        msg := priority-queue.Pull()   ▷ Fetch first message sorted by ts
18:        Trigger<p, self.topic, Deliver, msg>
19:        map = map /msg                                     ▷ Remove msg
20:        priority-queue = priority-queue /msg               ▷ Remove msg
```

Algorithm 1 makes atomic multicast possible for any event-source system. We have also proven that it is possible to implement a genuine Atomic Multicast system using an event-source system.

Simple Example

We assume there are 3 topics $\{topic_1, topic_2, topic_3\}$ denoted as Topic 1, Topic 2 and Topic 3 in Figure 3.3. We also assume there are three AtomicKafka nodes $\{node_1, node_2, node_3\}$, where each node subscribes to the corresponding topic number. When the message is sent to topic 1, we assume that the message is supposed to be AMCast to all three topics.

When AMCast is needed the process is always initiated by sending a ClientMessage to the Kafka cluster, this is the blue square in Figure 3.3. After the message has been handled by the AtomicKafka node the message will be forwarded as a NotifyMessage to the other two topics. This is shown as the red squares in Figure 3.3. The NotifyMessage is not sent to every topic as $topic_1$ does not notify itself, this happens because $topic_1$ was the receiver of the ClientMessage and already knows about it and immediately Acknowledges the message.

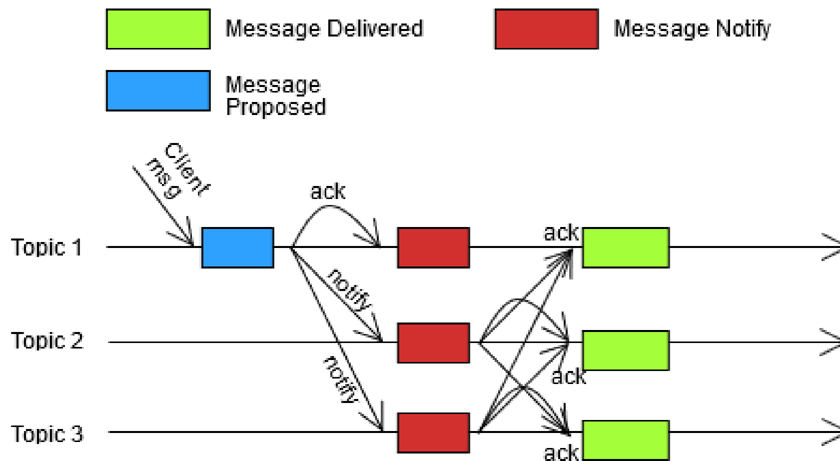

FIGURE 3.3: First design message propagation

When $topic_2$ and $topic_3$ receives the NotifyMessage they will process it according to the Algorithm 1 and reply with a response to the other topics. The topic that receives the ClientMessage $topic_1$ does not have to Acknowledge the message to the other nodes, as the NotifyMessage contains the information a Acknowledge message would contain, thus NotifyMessage and Acknowledge message is treated similarly.

After each node has received the Acknowledge messages they need they will do the last thing and decide on a message they will have to deliver which is shown as the green square in Figure 3.3.

Detailed Example

This example will give a more detailed insight into how the process works by using Figure 3.4. To simplify the example the empty spaces between the squares on the topic 1, 2, and 3 lines are assumed to be messages that do not belong to the AMCast between topic 1, 2, and 3.

We assume there are 3 topics $\{topic_1, topic_2, topic_3\}$ denoted as Topic 1, Topic 2 and Topic 3 in Figure 3.4. We also assume there are three AtomicKafka nodes $\{node_1, node_2, node_3\}$, where each node subscribes to the corresponding topic number. Two messages are sent to $topic_1$, message c and message b. Both messages are AMCasted from an AtomicKafka client. Using the Figure 3.4 we will go through the different phases and look at the messages being sent from each other, and how a latency issue will impact a topic.

In the top left corner of Figure 3.4 there are four rectangles with different colors, there are also three arrows with different colors. Each rectangle represents a different event in the Algorithm 1. The square represents the state the

message is currently in, and the arrows represent the control message being sent to the AtomicKafka nodes.

The light blue square named Message Proposed indicated that AtomicKafka node has received a ClientMessage from an AtomicKafka client, this is the method on Line method on Line 11. After a ClientMessage has been received it will then send NotifyMessages to the other nodes, represented by the blue arrow. When the nodes receive this NotifyMessage it will be represented as the red squares, named Notify Received, which is handled on Line 28. After the notification message has been processed it will respond with an AckMessage, represented by the purple arrow.

The dark blue square represents when a node has received all the AckMessages needed to make a decision, which is handled by the method on Line 5. This results in a message being able to be delivered, which is represented by the green square called Message Delivery.

We have simplified the data shown in the figure so information such as messageID and receivers has been removed. We kept the value information for the Message Proposed square but left out all other information. For the control messages, we have added a $m(x,y)$ where x is the value being AMCasted and y is the timestamp given to the message.

In figure 3.4 the first message proposed from a Atomickafka client is a message that contains the value "c" intended for $topic_1, topic_2, topic_3$. This is the first Step in the Figure 3.3.

As seen in the Algorithm 1 when a ClientMessage is sent to $topic_1$ and processed by $node_1$ the first thing the algorithm does is to assign the message a logical timestamp of 0, and then increases the timestamp by one in Line 12 and 13.

The Line 14 and 15 is used to build the state of AtomicKafka and a delivery order. In our example this map belonging to $node_1$ will eventually create a local message state overview of both $map := node_2, node_3$ for the AMCasted messages, where $node_2$ will have $map := node_1, node_3$ and $node_3$ has $map := node_1, node_2$.

Line 16 through 18 check if a message only has a single receiver, if yes it does not need to be sent to other nodes and can be delivered by the receiving node. In our example since there are two other receivers $topic_2, topic_3$ so $node_1$ has to send out a NotifyMessage to $topic_1, topic_2$. When $node_1$ is supposed to send the message to $topic_1$ it will instead update the message type to ACKNOWLEDGE and update the message state in the map, rather sending a message to the topic $node_1$ subscribes too. This can be seen in Figure 3.3

as the second communication step.

In Figure 3.4 when the message arrives in $topic_2, topic_3$ the node $node_2$, assuming there have been two message that is not shown, assigns 3 as the timestamp and $node_3$ has assigned the message 2 as the timestamp. This is done by the method on Line 28.

The LMSM of $node_2, node_3$ will look similar to this.

$$< 1, [\{2, < Acknowledge, "c" >\}, \{1, < Acknowledge, "c" >\}] >$$

.

After the NotifyMessage has been processed by $node_2, node_3$ both nodes will send an ACKNOWLEDGE message to each other. The particular case here is the AcknowledgeMessage from $node_2$ to topic $topic_3$ is delayed by some Δ time. The acknowledge phase is annotated as the second communication step in Figure 3.3.

At this moment $topic_1$ receives a new ClientMessage with value "a" and $node_1$ starts the notify phase again, giving the message a timestamp of 3. $node_2, node_3$ both receives the NotifyMessage and assigns timestamps of 5 and 7.

After $node_1, node_2$ has sent out the AcknowledgeMessage for the second message "a" both nodes have received all the AckMessage for message "c" to decide for delivery. As they compare the timestamps in their LMSM they both end up in this instance that "c" message has the highest timestamp of 3. As the message type is updated to a DeliveryMessage it is also reordered in the priority-queue and now the Delivery method in Algorithm 2 will run. Only $node_1, node_2$ will deliver the message with value "c" at this time to their own topics. This is the green square in Figure 3.3.

For $node_3$ the LMSM will look like

LISTING 3.3: LMSM composition

```
<messageID_1 ,
    [
        {senderID_1 ,timestamp_0 ,[topic_1,topic_2,topic_3] ,Acknowledge ," c " >} ,
        {senderID_3 ,timestamp_2 ,[topic_1,topic_2,topic_3] ,Acknowledge ," c "}
    ] ,
  messageID_2 ,
    [
        {senderID_1 ,timestamp_3 ,[topic_1,topic_2,topic_3] ,Acknowledge ," a "} ,
        {senderID_2 ,timestamp_5 ,[topic_1,topic_2,topic_3] ,Acknowledge ," a "} ,
```

```
            {senderID3 ,timestamp7 ,[topic1,topic2,topic3] ,Acknowledge ,  "a"}
        ]
    >
```

At this point $node_3$ has enough acknowledgments to make a decision and deliver the message "a" but is still missing an acknowledge from $node_2$ for message "c", meaning $node_3$ is not able to deliver the message "a" yet. While $node_1, node_2$ is deciding on the message "c" $node_3$ has received all the AcknowledgeMessages it needs to decide for the second message "a". This is handled by the RECEIVED-ACKNOWLEDGE method on Line 5 $node_3$ which decided the timestamp of the message should be 7 and is ready to be delivered. Since the message "a" is ready to be delivered the Delivery method in Algorithm 2 will decide that message "a" is not the message to be delivered first. This is because message "c" is already in the priority queue with a timestamp of 2 which has a higher priority as stated in Chapter 3.1.

After Δ time the last AcknowledgeMessage sent from $node_2$ arrives in topic $topic_3$ making it possible to make a decision on message "c". When the message "c" is updated with timestamp 3 both messages are ready to deliver. Meanwhile as both $topic_1, topic_2$ already contains the needed AcknowledgeMessages from $node_1, node_2, node_3$ both $node_1, node_2$ has decided to deliver the second message "a".

When the final message has been delivered all of the topics $topic_1, topic_2, topic_3$ will have the following message order c, b across the topics keeping the strict-ordering guarantee which we wanted to achieve.

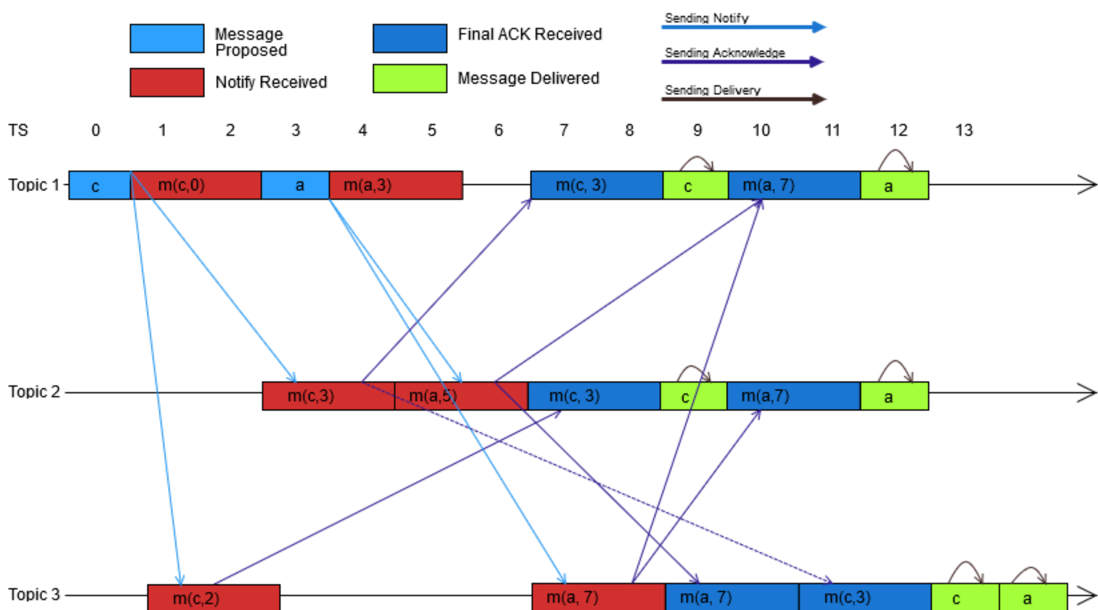

FIGURE 3.4: Example of phases and messages

3.1.2 Optimization

In the last chapter we discussed how we designed an algorithm solving the AMCast problem. We also explained the algorithm using an example proving that genuine Atomic Broadcast is possible when using an event-source system. As the algorithm was a Proof of Concept, we used a classical approach where every topic sends all messages between each other. By doing this we create $O(n^2)$ number of messages, and each node has to compute each message equally. The drawback by handling messages in this way is that if one topic receives more messages than others this node will delay the other nodes.

So in this chapter we will introduce an optimized AMCast suited for event-source systems. Where a single node is responsible for deciding if a message is deliverable or not. By doing this, we create a one-to-many and many-to-one communication rather than have many-to-many. Sending fewer messages overall.

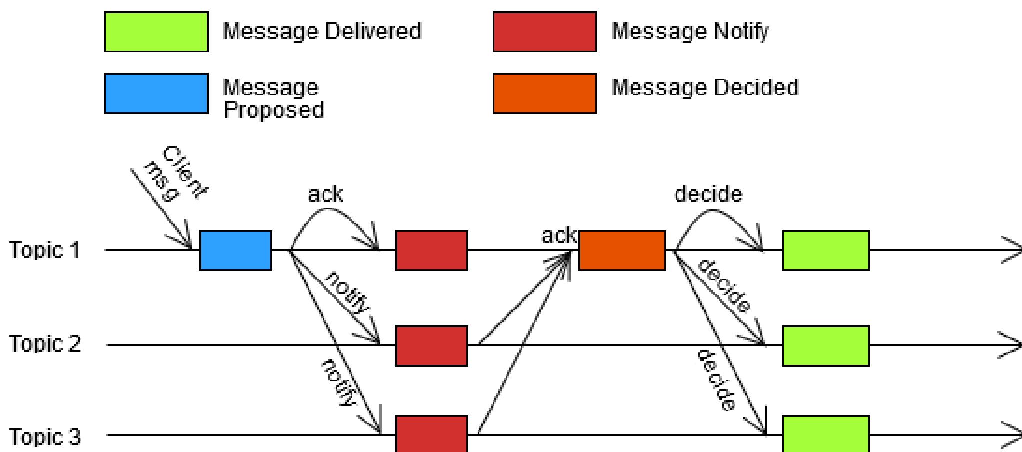

FIGURE 3.5: AMCast optimized message propagation

The Figure 3.5 shows how the optimized algorithm handles message propagation between the different topics. We have introduced a new communication step and a new message type decided. This allows a single node to be responsible for the messages it receives from a client and handle all the decision computation on a single node instead of every node. This design also adheres to the decoupled strategy we want to achieve, making each node more independent.

Simple Example

For this example we will be using Figure 3.5 to show how messages propagate through the system.

When topic 1 in Figure 3.3 when $topic_1$ receives a client message $\{node_1\}$ relays the message to $\{topic_2, topic_3\}$ intended receivers $topic_2, topic_3$. In the previous design both $\{node_2, node_3\}$ would send acknowledgment to the two other topics, in Figure 3.3 $\{node_2, node_3\}$ only needs to send back an acknowledge to $\{node_1\}$ instead. To enable this functionality only a single line of code was changed on line 38 in the Algorithm 3.

When $t1$ has received all the acknowledge messages from the other topics the received-acknowledge on Line 5 method is invoked. This method is only invoked by the nodes that received the first message from a client, shown in Figure 3.5. This method decides on a message in the same way as the previous PoC Algorithm 1. The difference between the methods in Algorithm 1 and 3 is that the latter algorithm is choosing a message which all of the nodes will agree on.

When $\{node_2, node_3\}$ receives the decided message the method in 4 on Line 18 will update the message state and the message reference in the heap queue.

After the heap queue has been updated it will trigger the method on Line 1 is invoked, which functions in the same way as the PoC algorithm 2 on Line 15.

Algorithm 3 AtomicKafka: Optimized part I

```
1:  Uses:
2:  Kafka-Producer instance p
3:  Kafka-Consumer instance c
4:
5:  on ⟨INIT⟩ do
6:      map := Π                                      ▷ Init message soft state map
7:      id := ℝ                                       ▷ The unique ID of the node
8:      priority-queue := Π ▷ Binary Heap used to sort deliverable messages
9:      timestamp := 0                                ▷ Lamport clock
10:
11: on event ⟨RECEIVED-CLIENTMESSAGE, msg⟩ do
12:     msg.timestamp = timestamp                     ▷ Add Lamport clock to message
13:     timestamp = timestamp +1                      ▷ Increase Lamport clock
14:     map = map ∪ msg                               ▷
15:     priority-queue = priority-queue ∪ msg
16:     if Σm.topics == 1∧ m.topics.contains(self.topic) then
17:         msg.type = Delivery
18:         priority-queue = priority-queue ∪ msg     ▷ Message is deliverable
19:     else                                          ▷ Find designated topics
20:         for topic ← msg.Topics do
21:             if topic == self.Topic then
22:                 msg.type = Acknowledge
23:                 priority-queue = priority-queue ∪ msg     ▷ Message is
    deliverable
24:             else
25:                 msg.type = Notify
26:                 Trigger <p, topic, Notify, msg>   ▷ Send to topics
27:
28: on event ⟨RECEIVED-NOTIFY, msg⟩ do
29:     msg.timestamp = timestamp
30:     timestamp = timestamp + 1
31:     map = map ∪ msg
32:     if Σm.topics == 1∧ m.topics.contains(self.topic) then
33:         msg.type = Delivery
34:         priority-queue = priority-queue ∪ msg     ▷ Message is deliverable
35:     else
36:         msg.type = Acknowledge
37:         priority-queue = priority-queue ∪ msg
38:         Trigger <p, msg.senderTopic, Acknowledge, msg>   ▷ Send ACKs
39:
```

Algorithm 4 AtomicKafka: Optimized part II

```
1:  Uses:
2:  Kafka-Producer instance p
3:  Kafka-Consumer instance c
4:
5:  on event ⟨RECEIVED-ACKNOWLEDGE, msg⟩ acknowledge from all do
6:      latestmsg := Π
7:      for storedMsg ← map[msg.id] do
8:          if storedMsg.ts > latestmsg.ts then        ▷ Finding biggest Timestamp
9:              latestmsg = storedMsg
10:     for topic ← msg.Topics do
11:         if topic != self.topic then                ▷ No need to send to self
12:             Trigger <p, topic, Decided, latestmsg>     ▷ Send Decided to topics
13:     latestmsg.type = Delivery
14:     priority-queue = priority-queue ∪ latestmsg    ▷ Update heap
15:     if timestamp < latestmsg.ts then
16:         timestamp = lastestmsg.ts
17:
18: on event ⟨DECIDED, msg⟩ do
19:     msg.type = Delivery
20:     priority-queue = priority-queue ∪ msg    ▷ insert decided msg into heap
21:     if timestamp < latestmsg.ts then
22:         timestamp = lastestmsg.ts
23:
24: on event ⟨DELIVERY⟩ inserted into priority-queue do
25:     While(priority-queue.peak() == Type.Delivery) then
26:         msg := priority-queue.Pull()    ▷ Fetch first message sorted by ts
27:         Trigger<p, self.topic, Delivery, msg>
28:         map = map /msg                            ▷ Remove msg
29:         priority-queue = priority-queue /msg      ▷ Remove msg
```

3.1.3 Enabling partitions for AtomicKafka

The previous Algorithm 1 and 3 are very generic in their design. Both algorithms would work with any event-source systems, and they only interact with messages and not the system itself. The algorithms allow us to enable AMCast on simple event-source systems, but is not optimal for advanced systems such as Kafka, as it does not fully utilize Kafka's potential by not being able to handle partitions. As explained in Chapter 2.4 Kafka is using partitions to increase the performance of the system. To fully utilize Kafka

we need to adapt our algorithm to conform to Kafka design by proposing a third design which utilizes partitions.

This third design is inspired by how Multi-Ring Paxos[18] handles multiple clustered Paxos instances similarly to how Kafka partitions the topics. Multi-Ring Paxos is an AMCast algorithm that is designed for scalability. The idea behind Multi-Ring Paxos is straightforward, by using what P. J. Marand et. al named Ring Paxos[19] one can initiate a consensus round. By using multiple rings they increase the throughput of Ring Paxos, but they have to be able to guarantee a total order. To guarantee total order Multi-Ring Paxos uses a merging function that can be solved deterministic, by combining the multiple individual rings into a single output at the end.

The merging can be very simplistic by using a round-robin fashion of taking the decided values from the first ring, then the second ring continuing onwards to the last ring before starting over at the first ring again.

As the Kafka partitions can be seen as rings in the sense of how P. J. Marand et al. defines a ring, we will create a merger like the one in Multi-Ring Paxos that can handle AMCast across topics. We already have developed an algorithm for AMcast across different topics we now have to design the system to keep strict ordering when a topic has been partitioned.

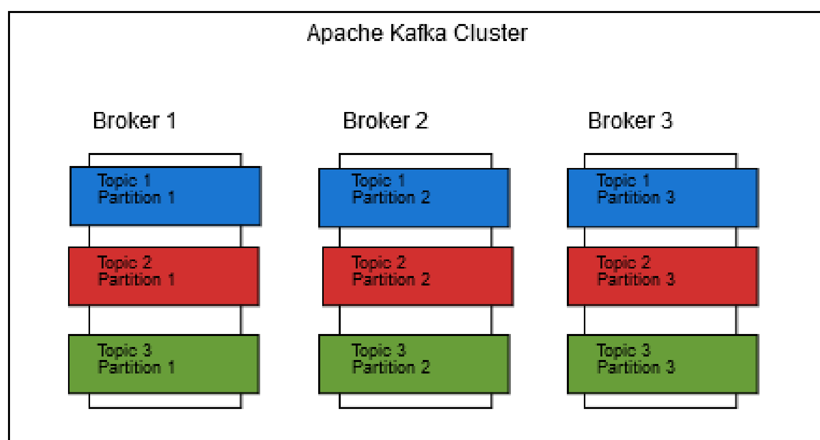

FIGURE 3.6: Topic Layering

Figure 3.6 show how a topic can be split into multiple partitions and reside in different brokers. In the figure all the topics are partitioned three times, though for end users it seems like a single topic, this is because the partitions together form a whole topic. In Figure 3.6 the topic partitions are group together by their partition number, this is what we call a partition layer. The grouped partition will never send control message or make

decisions for other partitions that do not have the same ID. The nodes that subscribe to the partitions in a partition layer belongs to a node layer.

If we assume that the topic 1 consist of $topic_1 := tp1_1, tp1_2, tp1_3$ and $topic_2 := tp2_1, tp2_2, tp2_3$ where tp is the partitions. When a node layer has decided on a message it will pass this to a merger which has the responsibility to deliver the messages in a sequence agreed on the partition layers, keeping the strict ordering. A partition layer consists of multiple partitions with identical partitionID, but not part of the same topic, this can be seen in Figure 3.6.

When a node layer has decided on a message, the decision is sent to the merger. The merger responsibility is then to ensure that the messages are delivered in the correct order based on the decision messages made by the partition layer nodes. 6 merges the individual partitions together.

This design does require more hardware if we use a single machine for each AtomicKafka process. This design will have $\mid tp \mid + \mid topics \mid$ machines running in the AtomicKafka cluster, where $\mid tp \mid$ is the total amount of partitions. Figure 3.7 shows how the setup is for a Cluster with $\mid topics \mid = 3$ where each topic is partitioned into three different partitions. This results in $\mid tp \mid + \mid topics \mid \longrightarrow 9 + 3 = 12$ machines.

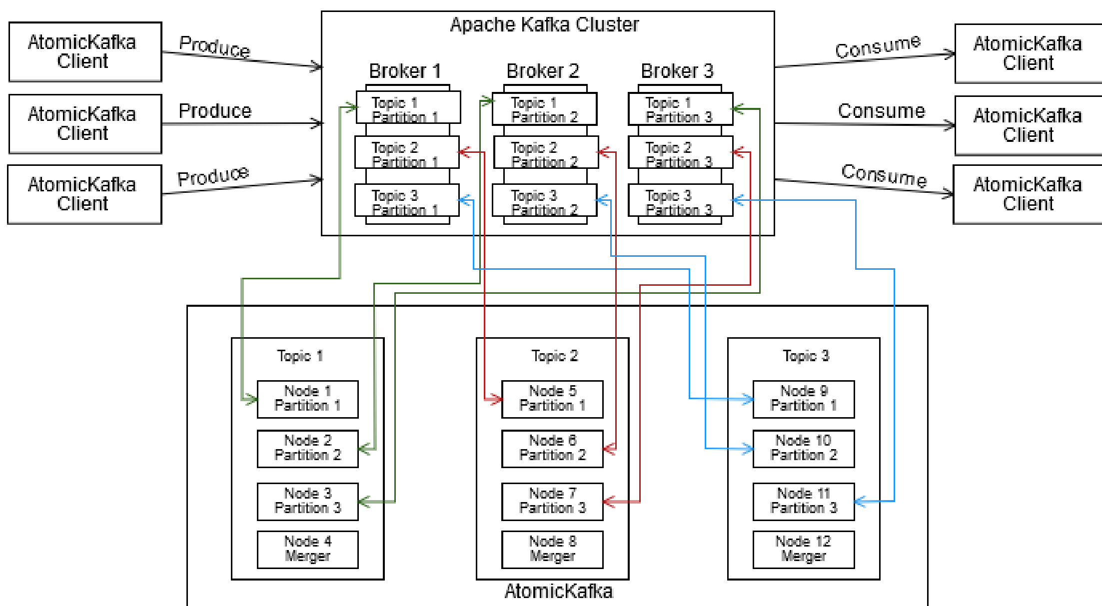

FIGURE 3.7: Partitioned AtomicKafka

To use the same example from Chapter 3.1, we need to add a new field called partitionID. The partitionID informs the nodes about what partition layer it belongs to, and can be seen in Listing 3.4. After the AMCast is done for the messages a snapshot of the priority-queue in the Merger algorithm

will look like Figure 3.7. The figure ends up looking the same because in the Merger the priority is as follows timestamp <= partitionID. If two messages have the identical timestamps AtomicKafka will use the partitionID as a tiebreaker. AtomicKafka uses partitionID and messageID as a tiebreaker since both will have some say in which messages were received first in correlation with the timestamp. Where the lowest msg with both partitionID and messageID will be prioritized.

We also have to introduce two new messages, the PartitionedClientMessage and Received-Partitioned-Delivery message.

LISTING 3.4: Message composition for partitions

$$msg_1 := < senderID_2, messageid_1, timestamp_1, partition_1, [topic_1, topic_2], ACKNOWLEDGE, "one" >$$

$$msg_2 := < senderID_2, messageid_2, timestamp_1, partition_2, [topic_1, topic_2, topic_3], ACKNOWLEDGE, "two" >$$

$$msg_3 := < senderID_2, messageid_3, timestamp_4, partition_3, [topic_1, topic_2, topic_3], ACKNOWLEDGE, "three" >$$

$$msg_4 := < senderID_2, messageid_4, timestamp_9, partition_1, [topic_1, topic_2, topic_3], ACKNOWLEDGE, "four" >$$

Merger

To prevent node layers from delivering DeliveryMessages directly to a user we have to modify the Delivery method in Algorithm 4. We need to define a new control message which allows the merges to know when a message is ready to be delivered. This modification is done in the algorithm below on Line 4.

Algorithm 5 AtomicKafka: Merger

1: **on** event ⟨DELIVERY⟩ inserted into priority-queue **do**
2:     **While**(priority-queue.peak() == Type.Delivery) **then**
3:         msg := priority-queue.Pull() ▷ Fetch first message sorted by ts
4:         **Trigger**<p, self.topic, PartitionDelivery, msg>
5:         map = map /*msg* ▷ Remove msg
6:         priority-queue = priority-queue /*msg* ▷ Remove msg

The merger process is initialized in Algorithm 6 on Line 5. In the initialization the merger is given an ID, the priority-queue which is the same kind of priority-queue mentioned in Chapter 3.1 with the modification of the prioritization mentioned above.

The hashmap which we refer to as LMSM does not exist in the merger, as it does not need to keep track of other merger nodes status and fully decoupled from other mergers.

In the initialization method on Line 5 the merger initial values are set up. Line 6 initialize the ID of the merger. Line 7 initialize the priority-queue, the priority-queue functions in the same way as the priority-queue in Algorithm 3. Line 8 initialize the lamport clock, and Line 9 initialize the lastDelivered variable that will contain the timestamp for the last delivered message.

When an AtomicKafka client sends a ClientMessage into the topic the merger registers the message on Line 11, as we explained in Chapter 2.4 the producer chooses the partition, and the merger does not need to worry about partitions. Line 12 creates a timestamp for the message, which will be used to position the message in priority-queue. After the message has received an timestamp, the lamport clock is increased by one on Line 13 and added to the priority-queue on Line 13.

When the merger is done with the Received-ClientMessage method it will wait for either for a new ClientMessage or that the node layer has made a decision returned by a PartitionDelivery message. When a PartitionDeliery message is sent into a topic the merger consumes and hands it to the method on Line 16. When the method is handed a message it will update the the message position in the queue based on the decision made by the node layer. The priority-queue will sort the messages based on the given timestamp, if there are two messages with the same timestamp the tiebreaker will be the one with the lowest partition as messages are delivered to the partitions in a round-robin style.

After the priority-queue has been updated the merger will see if there is any deliverable messages on Line 18. If there are any deliverable messages the algorithm will send a DeliveryMessage to the topic, remove the messages from the priority-queue and finally update the timestamps if the current delivered message has a higher timestamp than the local Lamport clock.

Algorithm 6 AtomicKafka: Merger

```
1:  Uses:
2:  Kafka-Producer instance p
3:  Kafka-Consumer instance c
4:
5:  on ⟨INIT⟩ do
6:      id := ℝ
7:      priority-queue := Π
8:      timestamp := 0
9:      latestDelivered := 0
10:
11: on event ⟨RECEIVED-CLIENTMESSAGE, msg⟩ do
12:     msg.timestamp = timestamp              ▷ Add Lamport clock to message
13:     timestamp = timestamp +1                       ▷ Increase Lamport clock
14:     priority-queue.add(msg)
15:
16: on event ⟨RECEIVED-PARTITIONDELIVERY, msg⟩ do
17:     priority-queue.update(msg)
18:     While(priority-queue.peak() == Type.PartitionDelivery) then
19:         msg := priority-queue.Pull()    ▷ Fetch first message sorted by ts
20:         Trigger<p, p.topic, Deliver, msg>
21:         latestDelivered := msg.ts
22:     if latestDelivered > timestamp then
23:         timestamp = lastDelivered + 1
```

3.2 Failure handling discussion

Failure handling is an essential aspect of a distributed system. If a process fails it could potentially halt the entire system and lose valuable information. To prevent this from happening to AtomicKafka it needs to have a failure-recovery strategy in place. This chapter we will discuss the possibility of rebuilding the LSMS and allowing the client to take over for failed nodes.

3.2.1 Rebuilding the local message state map

As AtomicKafka does not persist any information to a physical device, there is a possibility to retrieve all messages that have been previously processed by a node from Kafka. As mentioned in Chapter 2.4 a topic will retain a message for a certain period allowing us to do a replay of all the messages that have been processed by AtomicKafka.

Rebuilding the LMSM for an AtomicKafka node might be the simplest solution, though not without a cost.

Rebuilding LSMS: reading from beginning-to-end of topic

If one decides to rebuild the state by reading every message from beginning to the end, processing every control messages that exist in the subscribed topic, while rebuilding it should not be able to resend the messages, as it would end up violating the no duplication property. When every message has been processed it should rebroadcast the control messages missing a response. This is needed to ensure that the integrity property and total order property listed in Chapter 2.2 is kept.

Rebuilding LSMS: reading from end-to-beginning of topic

Reading from end to beginning is a more efficient approach as one could stop the reading process when all messages between the last DeliveryMessage, ClientMessage pair is consumed, minimizing the number of messages sent over the network. Stopping is possible as we know every timestamp preceding the last delivery is already delivered.

When all the messages have been added to the LSMS a scan for messages that already have been delivered can be initiated. If a delivered message is found all the control messages associated with that delivery can be deleted. After the scan, LSMS is ready and would be resending the control messages missing a response.

To do this the recovered process needs to consume the messages from Kafka from the end until it finds two delivery messages. Since a delivery message is the earliest to be delivered at that moment, meaning when we find the second last delivery message we know that every message that the node missed is being covered. The node can now safely start to replay messages from that point.

3.2.2 Client takeover

In a client takeover scenario a client would be a stateful client that also process the control messages to the topic it is subscribed too, without any influence in the decision making. This means that the node contains an LMSM and a priority-queue, and in the event of a node failure it could instantly take over for the failed node.

A stateful client would consist of the same modules as an AtomicKafka node, but would not actively engage in the decision making as there is already an AtomicKafka node in charge of the topic a stateful client would subscribe to.

In the event of a node becomes unresponsive a stateful client could take over from where the node left off after a specified timeout period. In the case of a stateful client take over the system would behave normally and not end up deadlocked.

When the failed node either becomes stable or comes back online it needs the ability to take back control over the topic from the client. There is also the issue if there are multiple clients, then we would have to implement a leader-election system and elect a leader between the clients.

The stateful client is not compatible with AtomicKafka if partitions are enabled, as a client treats the topic as a single topic and does not have any information about the partitions. To solve this issue we would have to resolve back to rebuilding the LSMS.

The issue with relying on only rebuilding a state is that if a node goes down, the system will have to wait until the state has been rebuilt. To overcome the issue of halting the system we propose to introduce a method of keeping states in clients, so in the event an AtomicKafka node fails a client could take over responsibility until either a new node has been introduced to the system or the node has recovered from the failure.

This functionality would complicate our algorithm but would allow $f < n - 1$ nodes to fail in theory, though with a high cost of throughput.

Chapter 4

Implementation

In this chapter we will explain how AtomicKafka was implemented in detail and discuss the architecture decisions we made. We will present the configurations we used for the Kafka producer and consumer in AtomicKafka so that the results of this thesis can be reproduced.

4.1 AtomicKafka

To confirm that atomic multicast is possible using Kafka we implemented Algorithm 1, and to evaluate the capabilities of AtomicKafka we implemented Algorithm 3. Our intention was to implement Algorithm 6, but because time constraints and issues described in Chapter 6.1.1 we had to drop the implementation of Algorithm 6.

The implementation of Algorithm 1 and Algorithm 3, hosted on GitHub[33], was implemented by using Java 8 and Kafka framework 2.0.0[23]. We also used other frameworks such as GSON 2.8.5[20] for JSON encoding/decoding, Metrics[21] for measuring performance and Apache Commons[22] for the priority queue implementation.

4.1.1 System Architecture

As building a distributed system can be complicated, we want to give some insight into the composition of AtomicKafka. First, we will write about the different classes and their relations, and then we will discuss how the Kafka consumer and Kafka producer was configured, as there are multiple configurations possible. Finally, we will discuss how we designed the messages used by AtomicKafka.

Figure 4.1 show a simplified class diagram of what has been implemented in AtomicKafka and the relation between the different classes.

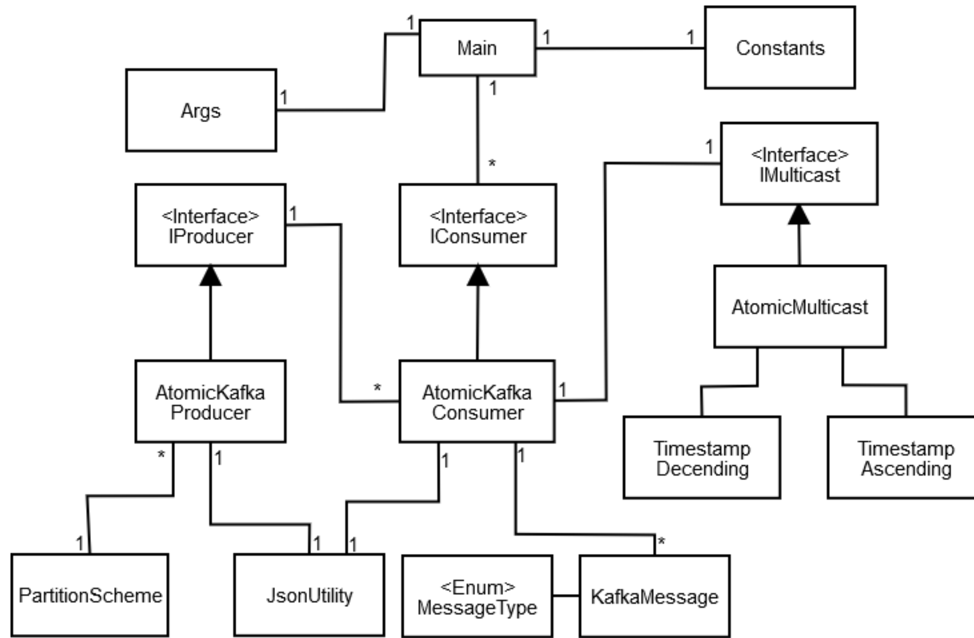

FIGURE 4.1: AtomicKafka Architecture

The main class has relation to three other classes; two of the classes are used to configure AtomicKafka at startup. The Constant class contains default information about broker addresses, topic name to subscribe too and ClientID, in case a user does not supply the application with startup arguments, while Args class is responsible for parsing arguments passed from console to the application. Each association (the edge line) is numbered by how many references a class can hold so that the main class can have one reference for Constant and one reference for the Arg class. The IConsumer interface reference in the main class states that main can have multiple instances of AtomicKafkaConsumer. This allows the main class to have multiple threaded instances of the AtomicKafkaConsumer class.

The intention of designing main to have multiple instances was to make it easier to implement the failure handling technique discussed in Chapter 3.2 in the future.

The AtomicKafkaConsumer class is responsible for consuming messages in a subscribed topic from the Kafka cluster, and then decode the consumed messages resulting in an AtomicKafkaConsumer object. To decode messages a JsonUtility instance is used, which is also referenced by the KafkaProducer class. When a message has been successfully decoded AtomicKafkaConsumer will identify what kind of message it is, as described in Chapter 4.1.4, by using the MessageType instance in KafkaMessage.

When AtomicKafkaConsumer has identified the message it will invoke a

method from the AtomicMulticast instance corresponding to the type. AtomicMulticast is the implementation of Algorithm 3. When AtomicMulticast has processed the message the result is then passed back to AtomicKafkaConsumer. If the result has to be sent back to the Kafka cluster, a method from AtomicKafka Producer is invoked.

There are two other classes AtomicMulticast has reference to, TimestampDecending and TimestampAscending, and the AtomicMulticast priority-queue uses these two classes. These two classes are used to define how elements being inserted into the queue are sorted.

The AtomicKafkaProducer is a wrapper for the Kafka frameworks producer class, and it is implemented as a singleton class, which multiple threads can use. It has been designed as a singleton class because the Kafka documentation states the following.

> "The producer is thread safe and sharing a single producer instance across threads will generally be faster than having multiple instances[14]."

The usage of AtomicKafkaProducer is to send a message from AtomicKafka to a specified topic residing inside the Kafka cluster. When sending a message AtomicKafkaProducer encodes it to the JSON format by using the JsonUltity class. AtomicKafkaProducer has a reference to PartitionScheme class for future implementation. The functionality of this class is described in Chapter 4.1.3.

4.1.2 Consumer

The Consumer client from the Kafka framework contains multiple different configurable parameters[13], some optional and some mandatory. The mandatory configurations have no default settings, meaning we have to supply information about broker address, what deserializer should be used for key and the value, and consumer ID. Changes we made to the optional configuration are shown in the Listing 4.1.

While we were researching how to optimize the algorithm, we thought of a use case that allows for better load balancing between the different nodes.

Enabling message forwarding to other topics, meaning that if an AtomicKafka client sends a message m to topic $topic_1$, but the message was intended for $topic_2$ the AtomicKafka node will forward the message to $topic_2$ instead of having the client resend the message itself. This allows a client to

use any topic it wants to send a message. Forwarding also enables developers to distribute the load more evenly across AtomicKafka in the case of fewer topics receives more traffic than others. The Algorithm 3 is designed to balance the load more evenly between the AtomicKafka nodes, by using nodes that have lesser load than others. Though this requires a load analysis on the Kafka topics to achieve maximum performance, which is out of the scope for this thesis.

LISTING 4.1: Consumer Configuration

```
"MAX_POLL_RECORDS_CONFIG" : "2000",
"ENABLE_AUTO_COMMIT_CONFIG : "true",
"BOOTSTRAP_SERVERS_CONFIG" : "host1:9092,host2:9092",
"KEY_SERIALIZER_CLASS_CONFIG" : LongSerializer.class.getName(),
"VALUE_SERIALIZER_CLASS_CONFIG" : StringSerializer.class.getName()
```

The bootstrap servers configuration is where we feed the broker addresses to the Kafka consumer. Having multiple addresses allows the consumer to connect to another broker in the list if a connection to the first broker cannot be established.

The max poll records configuration limits how many messages the consumer can consume from Kafka per pull request, meaning that if AtomicKafka has consumed 2000 messages it has to finish processing all the consumed messages before a new pull happens.

The auto-commit configuration allows the consumer to automatically tell Kafka that consumption has finished, allowing Kafka to know which messages have been consumed and which are to be consumed in the event of consumer failure.

The two serializers are used by Kafka to encode the information. The key serializer is set to the standard LongSerializer as we do not use the key for sending a message to Kafka, and the value serializer is set to StringSerializer as we encode our messages into JSON resulting in a string value.

4.1.3 Producer

The producer client from the Kafka framework also has configurable parameters, just like the Consumer client in Chapter 4.1.2. Just as we did with the

consumer client we decided to try and have as many default options as possible. The changes we made for the kafka producer is listed below in the Listing 4.2.

Configuration

LISTING 4.2: Producer Configuration

```
"BATCH_SIZE_CONFIG" : 0
"LINGER_MS_CONFIG" : 5
"ENABLE_IDEMPOTENCE_CONFIG" : "true"
"REQUEST_TIMEOUT_MS_CONFIG" : 30000
"BOOTSTRAP_SERVERS_CONFIG" : "host1:9092,host2:9092",
"KEY_SERIALIZER_CLASS_CONFIG" : LongSerializer.class.getName()
"VALUE_SERIALIZER_CLASS_CONFIG" : StringSerializer.class.getName()
```

When a producer is sending a message to Kafka, it can be either sent asynchronous or synchronous. Asynchronous send is a non-blocking method that offers better performance, while the synchronous send is a blocking operation. We discuss the challenges we faced with the asynchronous producer in Chapter 6.1.1.

Custom partitioning scheme

For enabling AtomicKafka partitions, as discussed in Chapter 3.1.3, we implemented a custom partitioner for the producer. The custom partitioner behaves similarly to the default one, explained in Chapter 2.4, meaning they both work in a round-robin[3] fashion when assigning a ClientMessage a partition. The issues are that AtomicKafka needs to be able to send the control messages to the same partitions, ensuring that the intended nodes receive each message.

Our partitioner works in a round-robin fashion when receiving a ClientMessage, assigning a different partition for each ClientMessage. When the AtomicKafka node needs to send a control message to the other topics it will read the partitionID from the message and set that ID as the partition target.

4.1.4 Messages

An AtomicKafka message, either control or delivery messages, differs from a normal Kafka message. Kafka messages is a key-value structure, where both

the key and value can be any Java Object defined by the developer[14]. For our purposes, the value section of the message will consist of an AtomicKafka message that is encoded as a JSON string. Chapter 3.1.3 will explain the details about how the key will be used to enable parallelism for AtomicKafka.

As seen in Listing 4.3, a message is transmitted by using JSON format. By using JSON format regular Kafka producer and Kafka consumer can be used. Regular consumers have a drawback as it would receive every message sent to a topic, including control messages meant for internal communication between AtomicKafka nodes. Either the regular consumer could filter out the messages not annotated as Delivery, or AtomicKafka could be configured to use two topics per intended topic, one for AtomicKafka internal communication and one topic for delivery messages. We have designed our system to use a single topic for all communication, though it would be trivial to change this.

4.1.5 Message Format

An AtomicKafka is defined as the listing 4.3. This listing shows an example of a message sent from a Client to the AtomicKafka cluster, where the only fields it manipulates is the MessageID, Topic and the value. The client will ignore all other fields as they are set and used by the AtomicKafka nodes.

```
1 {
2     "messageID": 1,
3     "senderID": 0,
4     "messageType": "ClientMessage",
5     "value": "Hello Kafka World",
6     "topic": ["T1","T2","T3","T4","T5"],
7 }
```

LISTING 4.3: AtomicKafka Client message example

Listing 4.4 shows an control message created as NotifyMessage in response to the incoming ClientMessage in Listing 4.3. This message used to notify the other topics about the ClientMessage. There is some added metadata on this message, such as which topic the message originated from, the nodeID of the sender and a timestamp. Message types are explained in chapter 2.4. The only fields that are changing in control messages are the timestamp, messageType, and SenderID.

```
{
    "messageID":1,
    "senderID":1,
    "sentFromTopic":"T1",
    "messageType":"NotifyMessage",
    "value":"Hello Kafka World",
    "topic":["T1","T2","T3","T4","T5"],
    "timeStamp":1
}
```

LISTING 4.4: Control message example

4.2 AtomicKafka Client

The AtomicKafka client was developed so that it filters out the control messages sent to a topic. Since we decided that DeliveryMessage and Client-, Notify-, Acknowledge- and DecidedMessage should all be sent to the same topic, we needed a client to filter out the delivered messages.

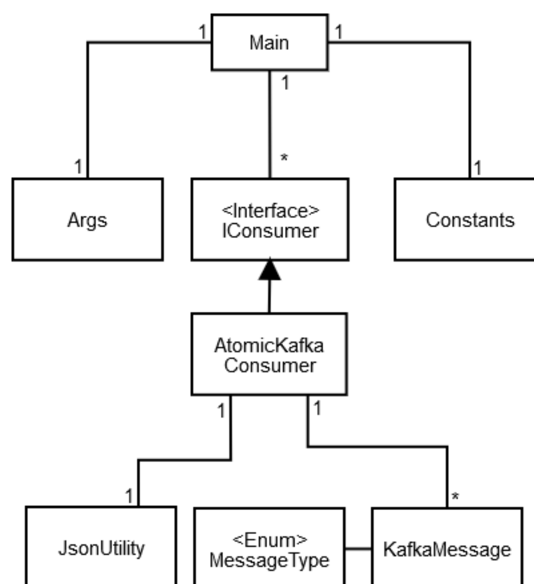

FIGURE 4.2: AtomicKafka Client Architecture

The Figure 4.2 resembles the AtomicKafka node architecture in Figure 4.1. This is because they are both based on the same framework, and we changed out the Consumer class with a simplified consumer, as explained above. The reason they are both built on the same framework is to make it possible to

implement the failure handling we discussed in Chapter 3.2, by replacing the consumer at runtime with an AtomicKafka node consumer.

Chapter 5

Evaluation

In this Chapter we will discuss the evaluation of AtomicKafka and what kind of hardware we used. We will present our final result of the evaluation. After presenting the result we will discuss the challenges we faced, potential improvements for future work and finally conclude our findings.

5.1 Hardware setup

The hardware we used to evaluate AtomicKafka we used the specified hardware in listing 5.1.

- Scientific linux 7.6 Kernel Version 3.10.0
- CPU 2.13 GHz Intel Xeon quad-core E5606
- RAM 16 GB
- Network Gigabit LAN environment
- Java SE Runtime Environment 1.8.3

The testing environment consists of twelve physically separated machines using the hardware listed above. These machines are not used exclusively by us and can be accessed by multiple people at the same time. To reduce the risks of outside influence on the test results we made sure that the evaluations ran at nighttime. At nighttime, there is less of a chance that people were using the machines. We also tried to make sure that no other demanding processes were running on the machine at the start of an evaluation. In the event someone would start a process while we were running the evaluations we had the processes distributed to separate machines, but we cannot guarantee how the network congestion was at the event of evaluation.

For the Kafka cluster, we use four machines, three of the machines were dedicated to three Kafka brokers, and the fourth is used as a Zookeeper node. Both broker nodes and the zookeeper node uses the default configuration, with some modification for the message retention period. The configurations are listed in the appendix.

For AtomicKafka, we use five machines, one for each topic, and we used two clients to produce the testing data. Lastly, we used a single regular Kafka consumer to read all messages going across one of the topics, and this was to ensure progress and to capture every message being sent to a topic and writes it to a file. In the event of failed evaluation run the stored file will allow us to analyze the evaluation run and see where the failure happened.

5.2 Evaluation setup

To evaluate AtomicKafka, we created five different topics inside the Kafka cluster, where each topic will have an AtomicKafka node subscribed to it. We will also use two AtomicKafka clients to send messages into the Kafka cluster and retrieve the delivered messages from AtomicKafka. Each AtomicKafka client will produce 20.000 messages each into Kafka, totaling 40.000 messages per evaluation scenario, where each AMCast message is formatted as shown in the Listing 4.3. As we had some issues with the Kafka producers, described in Chapter 6.1.1, we decided that the AtomicKafka clients should wait for 5ms after sending a message.

We have a total of five different testing scenarios where we try to simulate different load scenarios on AtomicKafka. We started by having a small load where 10% of the 20.000 messages sent from the producers are AMCast messages while the rest is only directed to the topic the producer is subscribed too. Each scenario increases this percentage by 10% until we reach approximately 50-50 mix of AMCast messages and messages being sent to a single topic.

To measure the performance we decided to measure three different parameters, how many incoming messages per second there are, outgoing second per minute and how many deliveries are happening per second. The two first measurements show us the throughput of the system, meaning each time a message is received or sent from an AtomicKafka node it is marked, then for each fifth second we calculate the mean value. The same mean value calculation is done for the delivery measurement, calculating how many DeliveryMessage a client receives per second.

To calculate the mean we are measuring how many messages have been registered per second, plotting each fifteen second into the graph.

5.3 Results

The results below are sectioned into six different sections with three different graphs and a table showing the average numbers. The graphs use two axes where the x-axis represents seconds, while the y-axis represents the number of messages.

5.3.1 Baseline

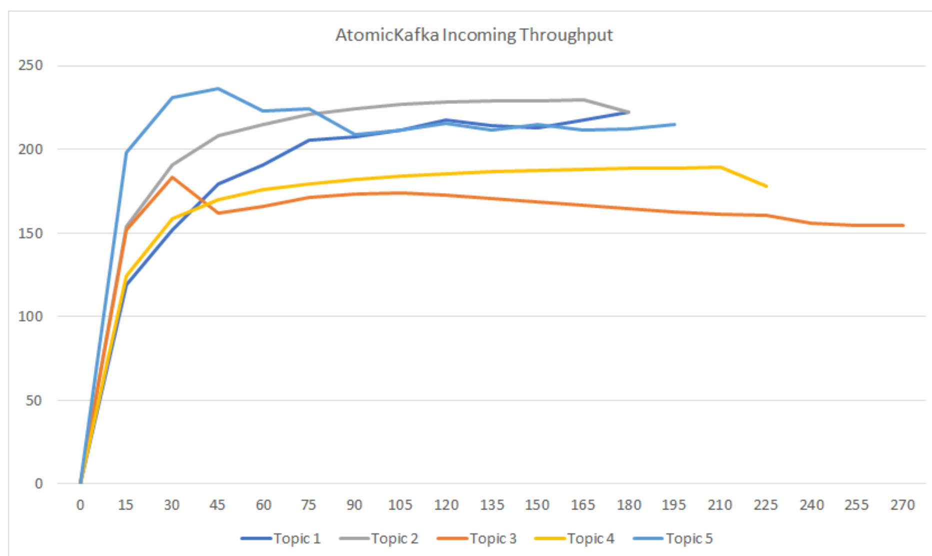

FIGURE 5.1: Baseline: Incoming messages per second

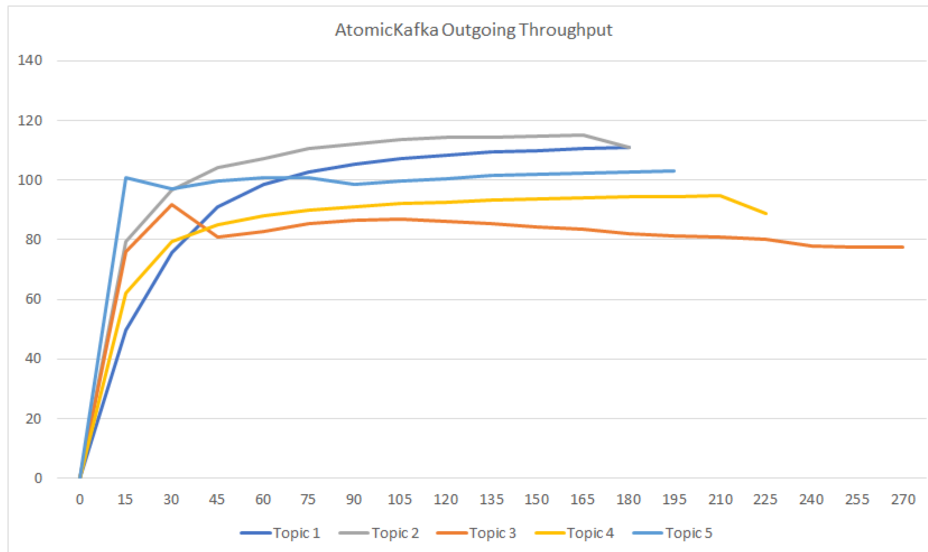

FIGURE 5.2: Baseline: Outgoing messages per second

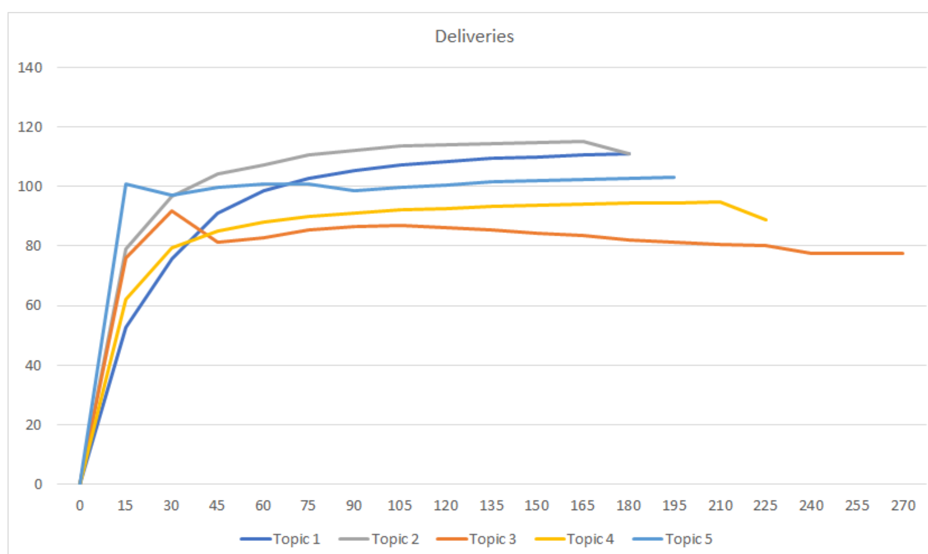

FIGURE 5.3: Baseline: Deliveries per second

TABLE 5.1: Baseline Averages

Node	Subscription	Incoming	Outgoing	Deliveries
Node 1	Topic 1	180.8 msg/s	90.7 msg/s	90.9 msg/s
Node 2	Topic 2	198.3 msg/s	99.4 msg/s	99.1 msg/s
Node 3	Topic 3	156.5 msg/s	78.3 msg/s	78.1 msg/s
Node 4	Topic 4	166.6 msg/s	83.3 msg/s	83.0 msg/s
Node 5	Topic 5	201.0 msg/s	93.5 msg/s	93.4 msg/s

The baseline evaluation shows us that some nodes are performing better than others. The Table 5.1 shows that node 2 has the best performance in the

evaluation, averaging ~198 messages per second for incoming, ~99 messages per second for outgoing and the client receiving on average ~99 delivered messages per second.

The worst performing node is node 3, having an average incoming rate of ~156 messages per second, ~78 outgoing per second, and the client receiving on average ~78 delivered messages per second.

The Figures 5.1, 5.2 and 5.3 displays that there is some startup latency when starting to congest messages, but stabilizes after a while. For the topic 3 we can see that it is performing worse than the other topics, indicating that the machine might struggle for resources.

The evaluation completed after 270 seconds. Sending and receiving 300000 messages in total.

5.3.2 90-10 Evaluation

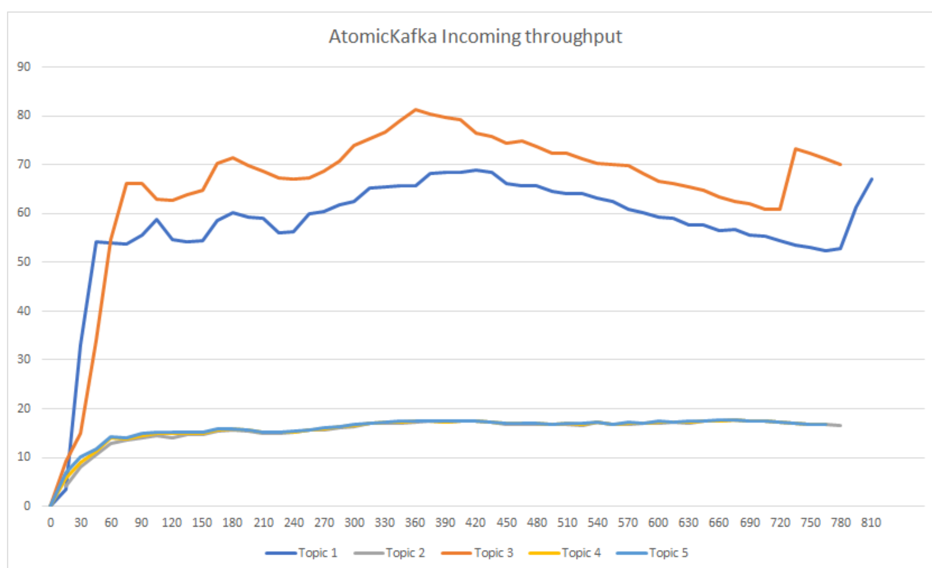

FIGURE 5.4: 10% AMCast: Incoming messages per second

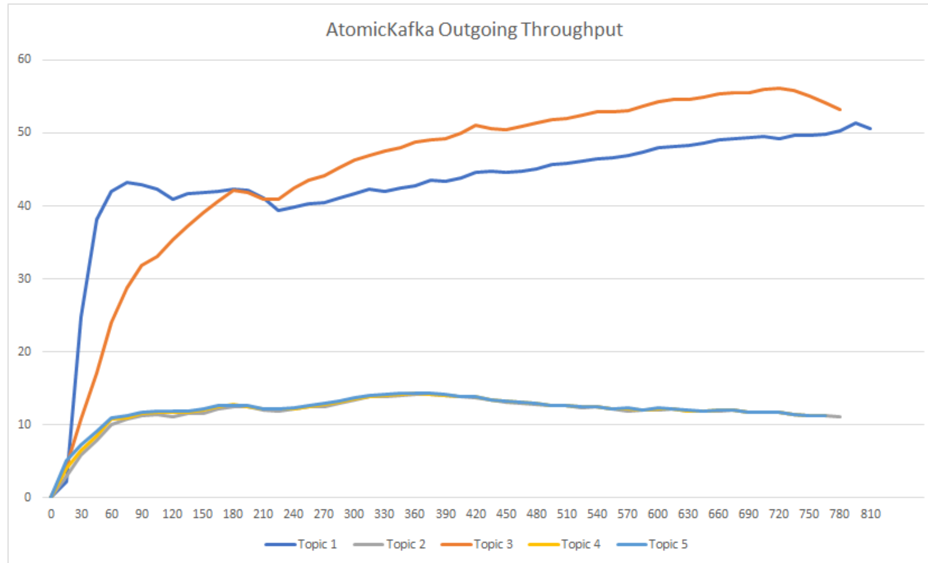

FIGURE 5.5: 10% AMCast: Outgoing per second

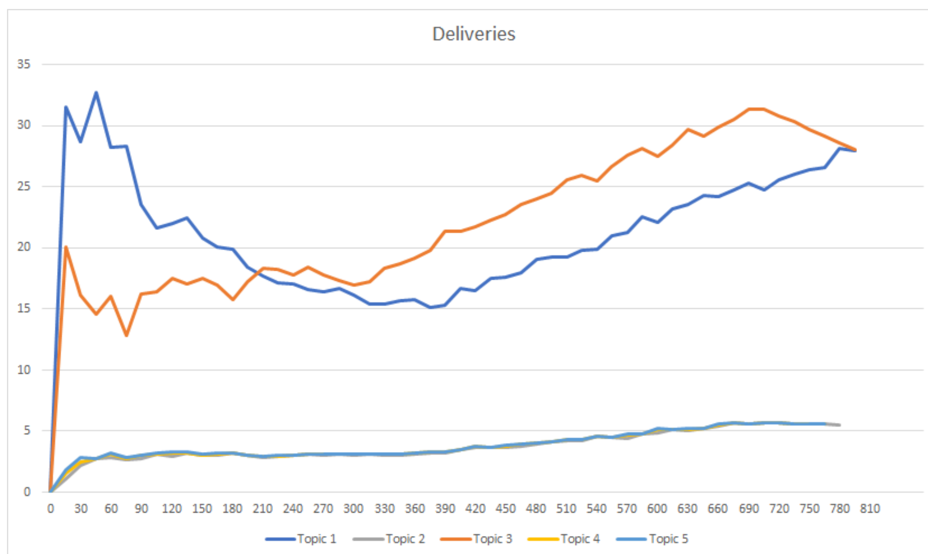

FIGURE 5.6: 10% AMCast: Deliveries per second

TABLE 5.2: 10% AMcast Averages

Node	Subscription	Incoming	Outgoing	Deliveries
Node 1	Topic 1	57.6 msg/s	42.9 msg/s	18.0 msg/s
Node 2	Topic 2	15.5 msg/s	11.7 msg/s	3.7 msg/s
Node 3	Topic 3	65.6 msg/s	44.6 msg/s	17.6 msg/s
Node 4	Topic 4	15.7 msg/s	11.9 msg/s	3.8 msg/s
Node 5	Topic 5	15.9 msg/s	12.1 msg/s	3.9 msg/s

From the Figure 5.4 it could seem like the node subscribed to topic 3 is performing better than every other node, but looking at Figure 5.5 and Figure 5.6 indicates that node 1 is busy delivering messages. The reason for topic

3 higher incoming messages rate, because node 3 has to wait for acknowledge messages from the other nodes, while topic 1 is delivering messages addressed to a single topic.

Table 5.2 shows a more accurate view of how each node was performing. As shown it node 1 and node 3 had the same performance numbers, and topic 2, 3, and 5 also had similar performance number. This is a good indication that there is no system falling behind on processing, though the graphs show that node 3 does need some time to catch up with node 1 as it received multiple single topic messages in the beginning.

Comparing to the baseline we see a large decrease in performance by a factor of ~5.05 for node 1 and ~4.43 for node 3, when comparing the Table 5.3 and Table 5.1.

We did not compare node 2,4, and 5 as their numbers would naturally increase with the amount of AMCast messages being sent to topic 2, 4, and 5.

The total run time was 810 seconds, using 310 seconds more than the baseline, sending and receiving 260883 messages in total.

5.3.3 80-20 Evaluation

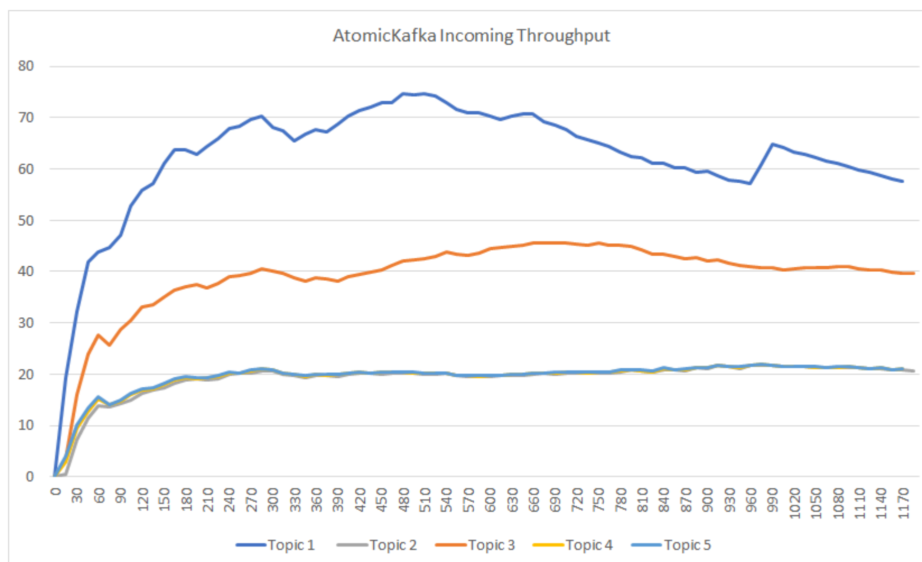

FIGURE 5.7: 20% AMCast: Incoming messages per second

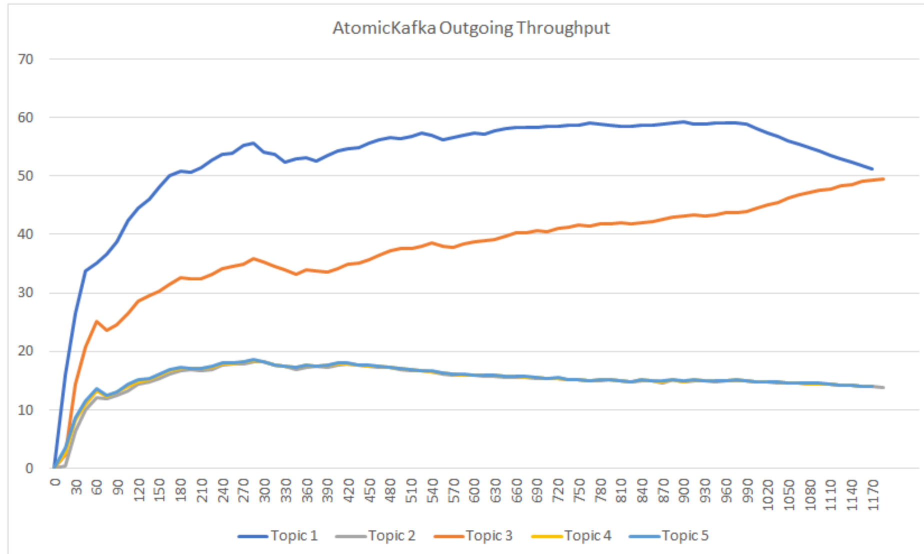

FIGURE 5.8: 20% AMCast: Outgoing messages per second

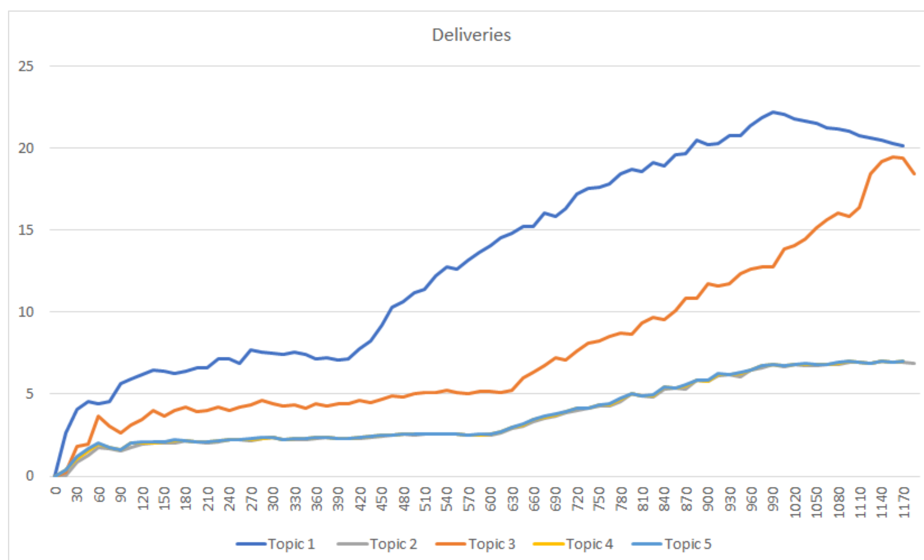

FIGURE 5.9: 20% AMCast: Deliveries per second

TABLE 5.3: 20% AMcast Averages

Node	Subscription	Incoming	Outgoing	Deliveries
Node 1	Topic 1	62.3 msg/s	52.9 msg/s	13.3 msg/s
Node 2	Topic 2	19.2 msg/s	15.0 msg/s	3.7 msg/s
Node 3	Topic 3	39.0 msg/s	37.3 msg/s	7.7 msg/s
Node 4	Topic 4	19.4 msg/s	15.3 msg/s	3.8 msg/s
Node 5	Topic 5	19.5 msg/s	15.3 msg/s	3.8 msg/s

The 20% AMCast evaluation sees a sudden drop in node 3 performance. Figure 5.7 shows that node 1 can consume more messages and send more

messages than node 3, which indicates the existence of a bottleneck. Because of this bottleneck, it is possible to assume that the other nodes are affected by this. Comparing the numbers between node 1 and node 3 in Table 5.3 shows a reduction in performance, while the other nodes have similar performance numbers.

Node 1 performances decreases by an factor of ~1.35, and node 3 decrease by an factor of ~2.28 for deliveries per second, when comparing the Table 5.3 and Table 5.2.

Comparing to the baseline we see a large decrease in performance by a factor of ~6.83 for node 1 and ~10.14 for node 3, when comparing the Table 5.3 and Table 5.1.

The total run time was 1185 and took 915 seconds more than the baseline. During the evaluation 361769 messages were sent and received in total.

5.3.4 70-30 Evaluation

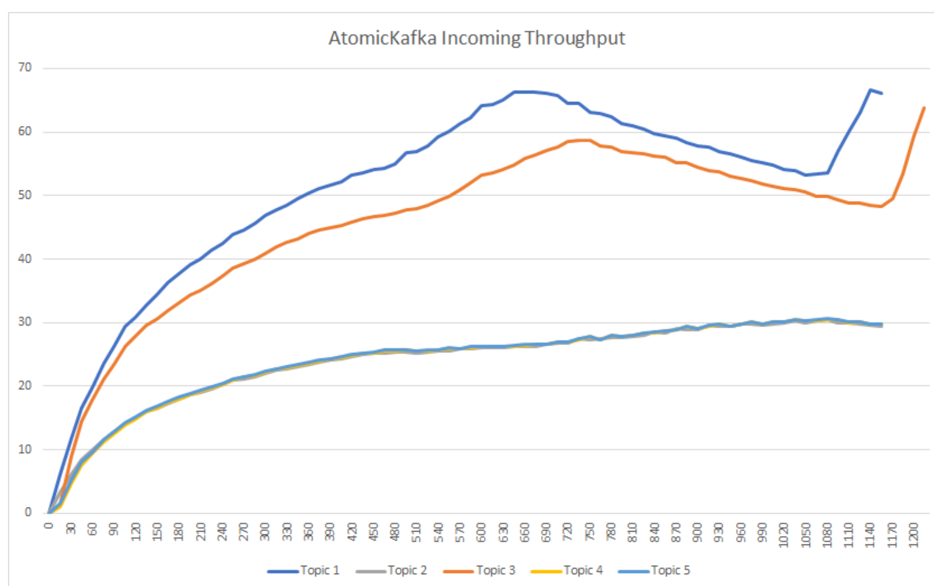

FIGURE 5.10: 30% AMCast: Incoming messages per second

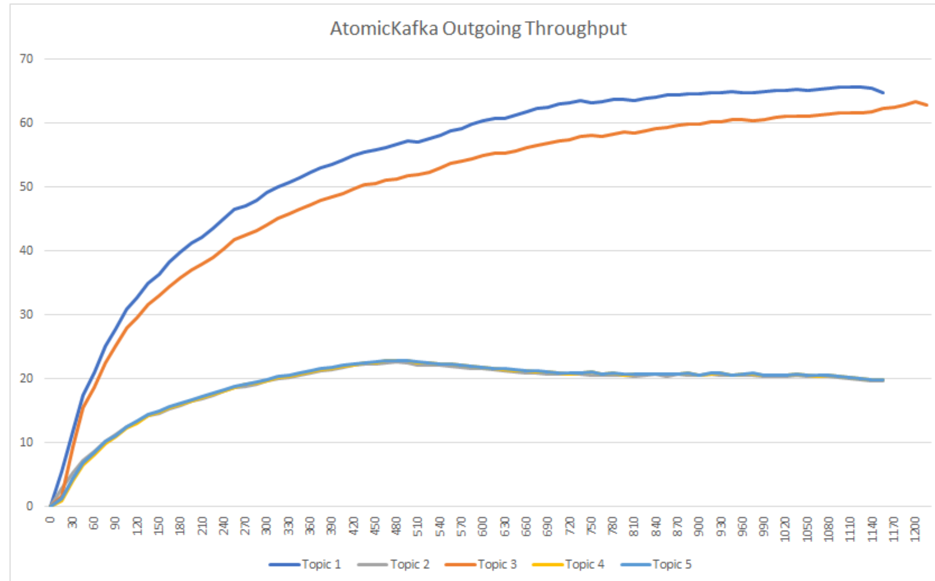

FIGURE 5.11: 30% AMCast: Outgoing messages per second

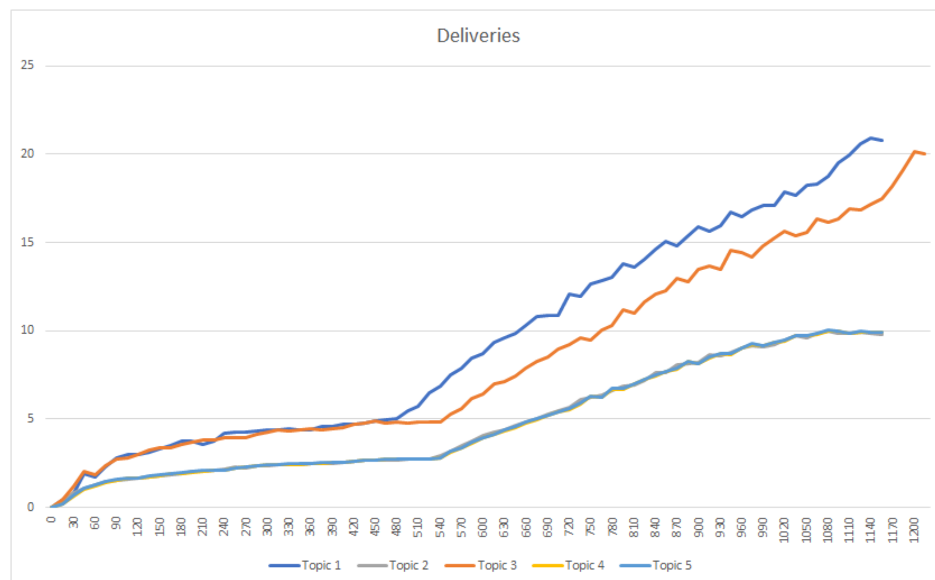

FIGURE 5.12: 30% AMCast: Deliveries per second

TABLE 5.4: 30% AMcast Averages

Node	Subscription	Incoming	Outgoing	Deliveries
Node 1	Topic 1	50.9 msg/s	53.5 msg/s	9.4 msg/s
Node 2	Topic 2	23.8 msg/s	18.8 msg/s	4.8 msg/s
Node 3	Topic 3	45.4 msg/s	49.6 msg/s	8.5 msg/s
Node 4	Topic 4	23.9 msg/s	18.9 msg/s	4.8 msg/s
Node 5	Topic 5	24.0 msg/s	19.0 msg/s	4.8 msg/s

For the 30% AMCast evaluation the Figure 5.10 and Figure 5.11 shows us node 3 still is underperforming compared to node 1, also confirmed by the

Table 5.4, though not as drastic as the graphs make it seem. Node 3 has 1 message less per second delivered on average than node 1, and this shows how vulnerable the system is to a bottleneck in a single node.

Figure 5.12 shows that when node 1 is starting to deliver more messages per second, topic 2, 3, 4 and 5 also start to deliver more messages, showing a correlation between message deliveries between the different nodes.

Node 1 performances decreases by an factor of ~1.41, and node 3 increases by an factor of ~0.90 for deliveries per second, when comparing the Table 5.4 and Table 5.3. Though the average differences seem small, they result in 180 seconds longer run time than the 30% AMCast evaluation.

Comparing to the baseline we see a large decrease in performance by a factor of ~6.71 for node 1 and ~10.14 for node 3, when comparing the Table 5.4 and Table 5.1.

Total run time was 1215 seconds, 945 seconds more than baseline. During the evaluation 515680 messages were sent and received in total.

5.3.5 60-40 Evaluation

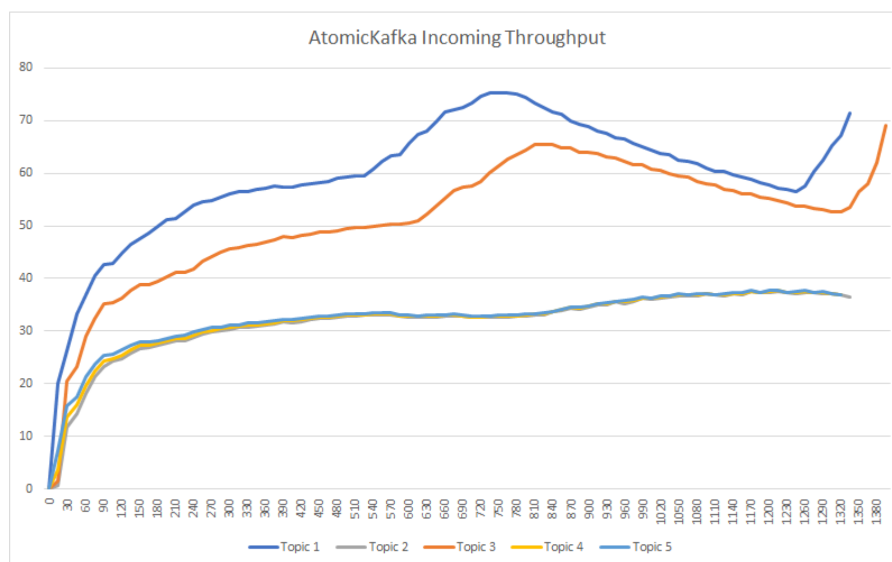

FIGURE 5.13: 40% AMCast: Incoming messages per second

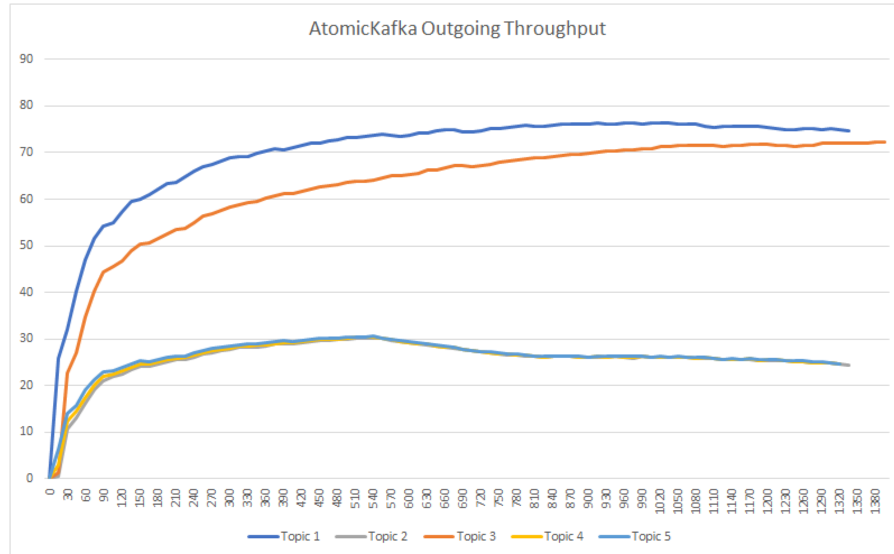

FIGURE 5.14: 40% AMCast: Outgoing messages per second

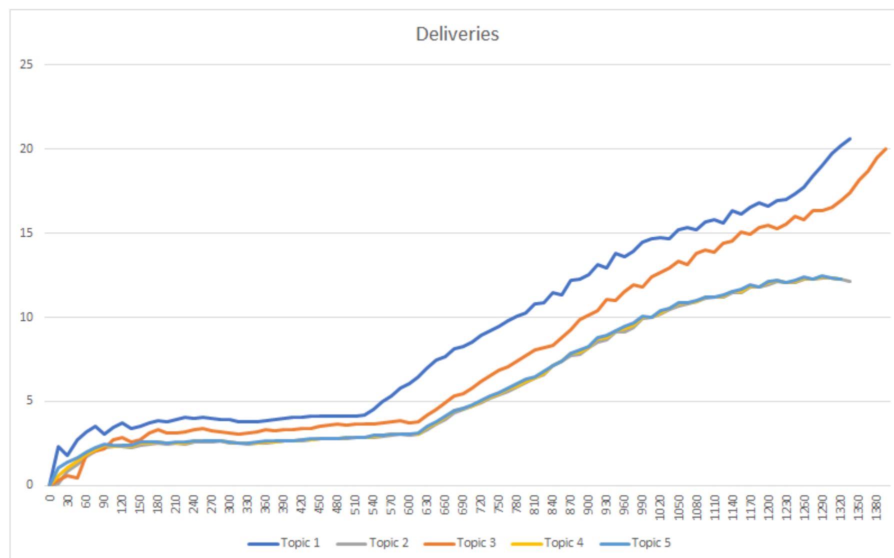

FIGURE 5.15: 40% AMCast: Deliveries per second

TABLE 5.5: 40% AMcast Averages

Node	Subscription	Incoming	Outgoing	Deliveries
Node 1	Topic 1	59.3 msg/s	69.7 msg/s	9.0 msg/s
Node 2	Topic 2	31.6 msg/s	25.5 msg/s	5.8 msg/s
Node 3	Topic 3	51.1 msg/s	62.4 msg/s	7.9 msg/s
Node 4	Topic 4	31.8 msg/s	25.8 msg/s	5.8 msg/s
Node 5	Topic 5	32.2 msg/s	26.1 msg/s	5.9 msg/s

The 40% AMCast evaluation averages in Table 5.5 shows that there is a slight decrease in delivered message per second for both node 1 and node

3, while node 2,4, and 5 has an increase, when comparing the averages to the 30% evaluation averages in Table 5.4. Node 1 performances decrease by a factor of 1.04, and node 3 decreases by a factor of 1.07 for deliveries per second, when comparing the Table 5.5 and Table 5.4 . Though the average differences seem small, they result in 180 seconds longer run time than the 30% AMCast evaluation.

Comparing to the baseline we see a large cost in performance by a factor of ~10 for node 1 and node 3, when comparing the Table 5.5 and Table 5.1.

The total run time was 1395 seconds, 1125 seconds more than the baseline. During the evaluation 644928 messages were sent and received in total.

5.3.6 50-50 Evaluation

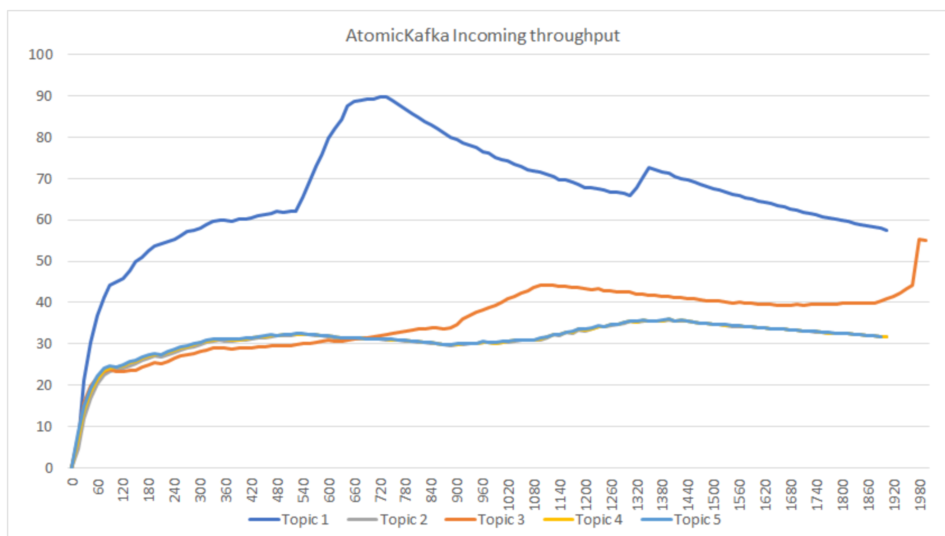

FIGURE 5.16: 50% AMCast: Incoming messages per second

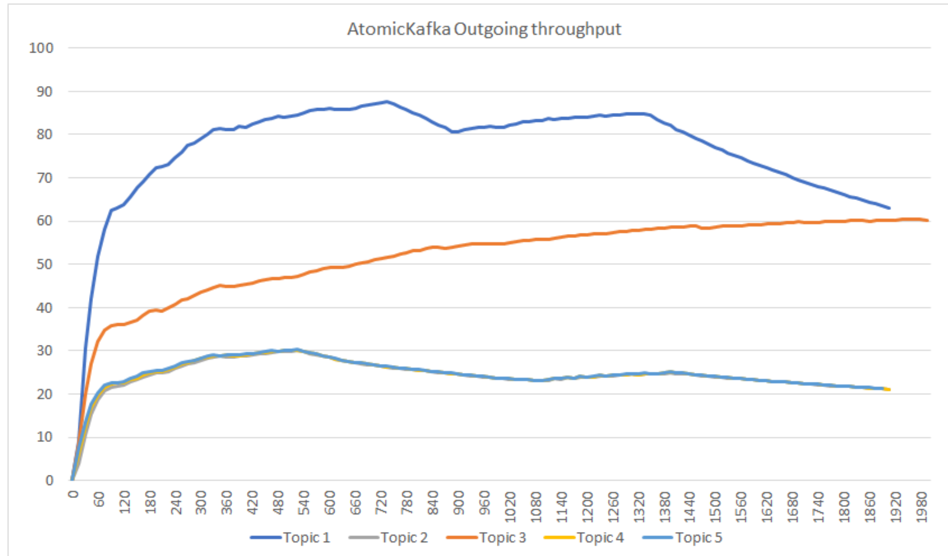

FIGURE 5.17: 50% AMCast: Outgoing messages per second

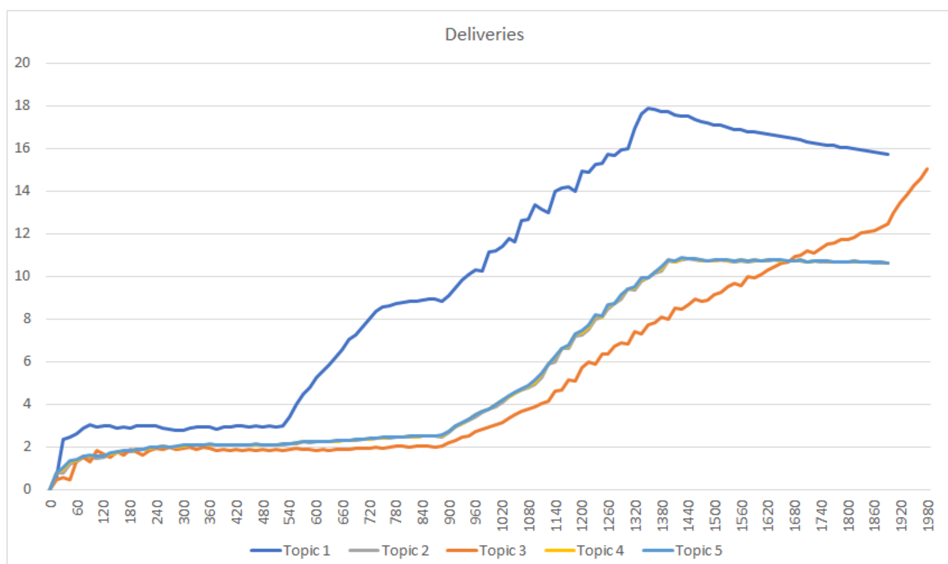

FIGURE 5.18: 50% AMCast: Deliveries per second

TABLE 5.6: 50% AMcast Averages

Node	Subscription	Incoming	Outgoing	Deliveries
Node 1	Topic 1	65.7 msg/s	76.5 msg/s	10.1 msg/s
Node 2	Topic 2	30.7 msg/s	24.3 msg/s	5.4 msg/s
Node 3	Topic 3	35.3 msg/s	51.6 msg/s	5.3 msg/s
Node 4	Topic 4	30.9 msg/s	24.9 msg/s	5.5 msg/s
Node 5	Topic 5	31.0 msg/s	24.6 msg/s	5.4 msg/s

For the 50% AMCast we see that node 1 performs significantly better than all the other nodes. According to Figure 5.16 node 1 can consume more messages than all other nodes, while node 3 performs worse than node 2, 4, and 5

at the start of the evaluation. This can also be seen in the deliveries in Figure 5.18 where node 3 bottlenecks the process, this assumption is also confirmed by looking at the averages from Table 5.6. The table indicates that node 3 is not able to keep up with the other nodes, and the Figures 5.16, 5.17, and 5.18 tells us that node 3 uses more time to finish delivering all the messages.

Node 1 performances increases by a factor of ~0.89, and node 3 decreases by a factor of ~1.49 for deliveries per second, when comparing the Table 5.6 and Table 5.5.

Comparing to the baseline we see a decrease in performance by a factor of 9.0 for node 1 and a decrease of ~14.73 for node 3 when comparing the Table 5.6 and Table 5.1.

Total run time was 1980 seconds, 1710 seconds more than the baseline, sending and receiving 771776 messages in total.

Chapter 6

Discussion and further work

In this chapter we discuss the evaluation results produced by AtomicKafka. We are also d

6.1 Discussion

The evaluations above indicate that there is a significant initial cost of introducing atomic multicast into Kafka. When we introduce 10% AMCast messages we see a significant decrease in throughput for both node 1 and node 3 in Figure 5.4 and Figure 5.5, the deliveries per second has the most significant reduction where node 1 is reduced from 90.9 messages per second down to 18.0 and node 3 goes from 78.1 messages per second down to 17.6.

Only looking at the rate of messages per second does not give us the whole picture of performance, time is also an important measurement. As with the decrease of end to end latency for the end client we see that the time spent to finish the evaluations also increases.

In the baseline evaluation the worst node took 270 seconds to complete, and when we introduced 10% AMCast messages the run time slowed down significantly by adding 310 seconds to the run time. That is an increase of 200% in time spent running the evaluation compared to the baseline run time. We knew that AtomicKafka would spend more time negotiating on the order of messages, but the drastic cost in time and the decrease of message throughput and the decrease in the number of messages delivered per second was not excepted to be so significant. We discuss the reason for the low performance and how it could be mitigated in Chapter 6.1.1.

The second round of evaluation with 20% AMCast messages ended up spending 46.2% more seconds than the 10% AMCast evaluation. That would indicate that each run time after 10% AMCast would increase by 46.2%, spending 375 seconds more to complete. We explained the reason for this in Chapter 5.3.3. While running the evaluation we were a significant performance

reduction on topic 3 that created a bottleneck slowing down the entire system. This is also confirmed by looking at the time spent when evaluating the 30% AMCast. In this case, the increase between 20% AMCast and 30% AMCast was only 2.5%. The 30% AMCast evaluation took 30 seconds more than the 20% AMCast evaluation. The suspected reason for this performance reduction for the 20% AMCast evaluation is part of the discussion in Chapter 6.1.1.

The 40% AMCast evaluation increases by 14.8%, spending 180 more seconds in time spent when comparing it to 30% AMCast runtime, which we think is a reasonable increase as we are increasing the AMCast messages by 10% and there is an additional cost as we are also increasing the number of control messages being sent through the system.

Finally, the 50% evaluation had an increase of 41.9%, 585 seconds more for the runtime when compared to the 40% evaluation, which is another significant increase in time.

What is promising about the differences in evaluation is that the differences between 30% AMCast and 40% AMCast shows an acceptable cost increase, and if it would be possible to replicate the 14.8% increase between every test it will show that the cost of increasing the amount of AMCast messages correlates with the time spent processing them making AtomicKafka cost efficient.

6.1.1 Performance

There are six obvious reasons for the performance reduction for throughput and end-to-end latency. The first reason is that we are using synchronized Kafka producers and not asynchronous producers. What this means is that for each message a producer sends a message it will block the thread until it receives an acknowledge from the broker that the message has been received, or it will retry to send the message after a timeout period. The reason we chose to use the synchronous was because of an issue with Kafka losing messages while using the asynchronous producer. Due to time constraints and already having spent much of our time problem solving this issue we decided to use instead synchronous producers which did not lose any messages.

The second reason is that we are not batching the messages together when we are sending, we are sending each message individually. According to the documentation[12] batching small messages together is encouraged as it

boosts the performance of Kafka significantly. The reasons for not batching was to simplify the debugging process while developing.

The third reason is how the Kafka consumer pulls messages from the Kafka cluster. Currently, the consumer will block processing the messages until it has either waited for one second or when the Kafka consumer has pulled 500 messages. These parameters could be tweaked to allow faster consumption increasing the incoming throughput, allowing for a faster consensus as more messages are consumed, and the nodes will not have to wait for too long for the control messages needed to deliver a message.

The fourth reason is using the same topic for control messages and deliveries, polluting the topic. Since AtomicKafka needs control messages to be able to make a decision on when to deliver new messages, reading already delivered messages slow this process down. Currently, if we are sending x messages we know that there will be at least 2x messages consumed. This is somewhat a waste of resources as each delivered message has to be consumed from the Kafka cluster then parsed and finally be discarded. The reason we chose to use a single topic for both control messages and delivery messages is because of AtomicKafka ability to be integrated with existing Kafka clusters. By using a single topic we know we get lower throughput and end to end latency, but using a single topic for both kinds of messages creates a better picture if one wants to integrate instead for creating a new cluster, as the performance will increase.

The fifth reason is unstable machines. By looking at the Figures 5.1, 5.2, and 5.1 there is a discrepancy between the different nodes even though they are on the same type of hardware and the same network as each other and the Kafka clusters. A potential reason for this discrepancy could be because of the shared machines, as mentioned in Chapter 5.1. The shared machines might be one of the reasons we see a performance drop for the 20% evaluation.

The five issues above only solve performance issues for AtomicKafka itself; there is also the possibility to increase the Kafka clusters performance. As we stated in Chapter 5.1 the default settings for the Kafka cluster was used, as optimization for the Kafka cluster is out of scope for this thesis.

The sixth reason is how often we check for deliverable messages. Currently, the AtomicKafka implementation checks for deliverable messages after all messages have been processed, this was chosen to make debugging easier when implementing the algorithms.

If one would be able to overcome these issues discussed above there is

potentially a significant increase in performance for AtomicKafka, making AtomicKafka desirable for service providers such as Netflix, as we mentioned in Chapter 1.

Our biggest challenge during the implementation was finding the problem with missing messages, because of the complexity of Kafka and our implementation there could be many reasons for missing messages. After spending multiple days of debugging we found that the synchronous producer did not exhibit these issues. Because of the time spent trying to find and solve the issue with missing messages we were not able to implement the partitioning of AtomicKafka.

6.1.2 Further work

As we mentioned in Chapter 6.1.1 there are many changes to both AtomicKafka and Kafka itself to increase the performances.

- Asynchronous Producer: There is a significant cost of using the synchronous producer, as it has to wait for an acknowledge from the broker that a message has been received. Solving this issue could increase the throughput for the whole system and thus increase the overall performance of the system.
- Batching: Implementing batching could reduce congestion of the network, but increase in bandwidth, also allowing for more messages to pass through Kafka at once. As we stated above in Chapter 6.1.1 batching is recommended by the Kafka documentation.
- Compression: If batching is enabled there could also be a benefit of using compression[13] on the batch, this is also recommended by the Kafka documentation.
- Separation of control messages and deliveries: The possibility of throughput suffering from having delivered messages in the same topic as all the other control messages should be evaluated, and considered to implement a mechanism that allows the existence of both mechanisms.
- Delivery check mechanism: An evaluation should be conducted to see how often AtomicKafka should check for deliverable messages, should it be triggered to see if there are any deliveries, should it be checked based on a timed function or checked for every message received.

- Enabling partitioning for AtomicKafka: Enabling partitioning for AtomicKafka would allow it to interact with Kafka clusters using partitions already, and could potentially increase the throughput for the whole system as it would adapt to Kafka's design.
- Failure handling: As we stated in Chapter 3.2, failure handling was out of scope for this thesis, but to be able to be used in a real-world scenario one would need to implement failure handling.

Chapter 7

Conclusion

This thesis has explored the possibility of introducing atomic multicast to an event-source system called Kafka. We have presented a proof of concept algorithm that shows atomic multicast is possible on a event-source system. We also present an optimized algorithm based on the proof of concept and an algorithm that can handle topics that are partitioned. We have also presented an implementation of the optimized algorithm in a system called AtomicKafka, which is a modular extension to Kafka.

Our results show that using atomic multicast with Kafka has good potential to be used in the industry if the implementation is done correctly. The evaluation was conducted by sending a total of 40.000 messages from two producers, and we achieved a baseline where the client was able to consume messages at a max average rate of 93.4 messages per second. The first evaluation atomic multicast %10 of those 40.000 messages and was able to achieve max average rate of 18.0 messages per second. However, because of performance deviations in the result, it is hard to conclude that this is the best performance we can achieve, as the implementation could be optimized and the test systems could be stabler. The results shows that atomic multicast is possible on event-source system, and with some optimization it can also be cost efficient.

Appendix A

Program Code

The full program code for the implementation is under a version controlled repository that was used in the development process.

Repository: https://github.com/destidom/atomickafka.

Appendix B

Kafka Broker Configuration

```
#### Server Basics ####
# The id of the broker. This must be set to a unique
   ↪ integer for each broker.
broker.id=1
listeners=PLAINTEXT://:9001
log.dirs=/home/stud/<username>/kafka/kafka_2.11-2.0.0/
   ↪ tmp/kafka-logs-0
#### Socket Server Settings ####

advertised.listeners=PLAINTEXT://<URL>:9001
# The number of threads that the server uses for
   ↪ receiving requests from the network and sending
   ↪ responses to the network
num.network.threads=3

# The number of threads that the server uses for
   ↪ processing requests, which may include disk I/O
num.io.threads=8

# The send buffer (SO_SNDBUF) used by the socket
   ↪ server
socket.send.buffer.bytes=102400

# The receive buffer (SORCVBUF) used by the socket
   ↪ server
socket.receive.buffer.bytes=102400

# The maximum size of a request that the socket server
   ↪  will accept (protection against OOM)
```

```
socket.request.max.bytes=104857600


#### Log Basics ####

# A comma separated list of directories under which to
   ↪  store log files
log.dirs=/home/stud/<username>/kafka/kafka_2.11-2.0.0/
   ↪ tmp/kafka1

# The default number of log partitions per topic. More
   ↪  partitions allow greater
# parallelism for consumption, but this will also
   ↪ result in more files across
# the brokers.
num.partitions=1

# The number of threads per data directory to be used
   ↪ for log recovery at startup and flushing at
   ↪ shutdown.
# This value is recommended to be increased for
   ↪ installations with data dirs located in RAID
   ↪ array.
num.recovery.threads.per.data.dir=1

#### Internal Topic Settings####
# The replication factor for the group metadata
   ↪ internal topics "__consumer_offsets" and "
   ↪ __transaction_state"
# For anything other than development testing, a value
   ↪  greater than 1 is recommended for to ensure
   ↪ availability such as 3.
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min.isr=1

#### Log Flush Policy####
```

```
# The number of messages to accept before forcing a
   ↪ flush of data to disk
log.flush.interval.messages=300000

# The maximum amount of time a message can sit in a
   ↪ log before we force a flush
#log.flush.interval.ms=1000

#### Log Retention Policy ####

# The following configurations control the disposal of
   ↪  log segments. The policy can
# be set to delete segments after a period of time, or
   ↪  after a given size has accumulated.
# A segment will be deleted whenever *either* of these
   ↪  criteria are met. Deletion always happens
# from the end of the log.

# The minimum age of a log file to be eligible for
   ↪ deletion due to age
log.retention.hours=24

# A size-based retention policy for logs. Segments are
   ↪  pruned from the log unless the remaining
# segments drop below log.retention.bytes. Functions
   ↪ independently of log.retention.hours.
#log.retention.bytes=1073741824

# The maximum size of a log segment file. When this
   ↪ size is reached a new log segment will be created
   ↪ .
log.segment.bytes=1073741824

# The interval at which log segments are checked to
   ↪ see if they can be deleted according
# to the retention policies
log.retention.check.interval.ms=300000
```

```
#### Zookeeper ####

# Zookeeper connection string (see zookeeper docs for
   ↪ details).
zookeeper.connect=<URL>:2181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=6000

#### Group Coordinator Settings ####
group.initial.rebalance.delay.ms=0

#### other ####
delete.topic.enable=true
log.cleanup.policy=delete
```

Bibliography

[1] Apache. *Apache Kafka current version*. 2019. URL: https://github.com/apache/kafka/tree/407bcdf78e06f83f2b358d2cbd96aed348a5c28f.

[2] Apache. *Apache Kafka repository*. 2019. URL: https://github.com/apache/kafka.

[3] Apache. *apache/kafka/DefaultPartitioner.java*. URL: https://github.com/apache/kafka/blob/trunk/clients/src/main/java/org/apache/kafka/clients/producer/internals/DefaultPartitioner.java.

[4] S. Benz and F. Pedone. "Elastic Paxos: A Dynamic Atomic Multicast Protocol". In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 2157–2164.

[5] N. T. Blog and N. T. Blog. *Auto Scaling Production Services on Titus*. 2018. URL: https://medium.com/netflix-techblog/auto-scaling-production-services-on-titus-1f3cd49f5cd7.

[6] N. T. Blog and N. T. Blog. *Kafka Inside Keystone Pipeline*. 2016. URL: https://medium.com/netflix-techblog/kafka-inside-keystone-pipeline-dd5aeabaf6bb.

[7] N. T. Blog and N. T. Blog. *Netflix Conductor: A microservices orchestrator*. 2016. URL: https://medium.com/netflix-techblog/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40.

[8] N. T. Blog and N. T. Blog. *NTS: Real-time Streaming for Test Automation*. 2015. URL: https://medium.com/netflix-techblog/nts-real-time-streaming-for-test-automation-7cb000e933a1.

[9] P. R. Coelho, N. Schiper, and F. Pedone. "Fast Atomic Multicast". In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2017, pp. 37–48. DOI: 10.1109/DSN.2017.15.

[10] G. DeCandia et al. "Dynamo: amazon's highly available key-value store". In: *ACM SIGOPS operating systems review*. Vol. 41. 6. ACM. 2007, pp. 205–220.

[11] M. J. Fischer. "The consensus problem in unreliable distributed systems (a brief survey)". In: *International conference on fundamentals of computation theory*. Springer. 1983, pp. 127–140.

[12] A. S. Foundation. *A Guide to the Kafka Protocol*. 2018. URL: https://kafka.apache.org/protocol.html.

[13] A. S. Foundation. *Apache Kafka Documentation*. 2018. URL: https://kafka.apache.org/documentation/.

[14] A. S. Foundation. *Apache Kafka KafkaProducer*. 2018. URL: https://kafka.apache.org/090/javadoc/index.html?org/apache/kafka/clients/producer/KafkaProducer.html.

[15] R. Guerraoui and A. Schiper. "Genuine atomic multicast in asynchronous distributed systems". In: *Theoretical Computer Science* 254.1-2 (2001), pp. 297–316.

[16] L. Jehl. "Decoupling atomic multicast". In: University of Stavanger. 2018.

[17] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (1978), pp. 558–565.

[18] P. J. Marandi, M. Primi, and F. Pedone. "Multi-Ring Paxos". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 2012, pp. 1–12. DOI: 10.1109/DSN.2012.6263916.

[19] P. J. Marandi et al. "Ring Paxos: A high-throughput atomic broadcast protocol". In: *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*. IEEE. 2010, pp. 527–536.

[20] *Maven Repository: com.google.code.gson » gson » 2.8.5*. URL: https://mvnrepository.com/artifact/com.google.code.gson/gson/2.8.5.

[21] *Maven Repository: io.dropwizard.metrics » metrics-jmx » 4.1.0*. URL: https://mvnrepository.com/artifact/io.dropwizard.metrics/metrics-jmx/4.1.0.

[22] *Maven Repository: org.apache.commons » commons-lang3 » 3.0*. URL: https://mvnrepository.com/artifact/org.apache.commons/commons-lang3/3.0.

[23] *Maven Repository: org.apache.kafka » kafka-clients » 2.0.0*. URL: https://mvnrepository.com/artifact/org.apache.kafka/kafka-clients/2.0.0.

[24] N. Narkhede, G. Shapira, and T. Palino. *Kafka: The Definitive Guide Real-Time Data and Stream Processing at Scale*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491936169, 9781491936160.

[25] A. B.F.D.F. P. Paulo Coelho Tarcisio Ceolin Junior. "Byzantine Fault-Tolerant Atomic Multicast". In: 2018. URL: http://www.di.fc.ul.pt/~bessani/publications/dsn18-byzcast.pdf.

[26] J. Rao. *How to choose the number of topics/partitions in a Kafka cluster?* 2018. URL: https://www.confluent.io/blog/how-choose-number-topics-partitions-kafka-cluster.

[27] J. Rao. *Open-sourcing Kafka, LinkedIn's distributed message queue*. 2011. URL: https://blog.linkedin.com/2011/01/11/open-source-linkedin-kafka.

[28] L. Rodrigues, R. Guerraoui, and A. Schiper. "Scalable atomic multicast". In: *Proceedings 7th International Conference on Computer Communications and Networks (Cat. No. 98EX226)*. IEEE. 1998, pp. 840–847.

[29] Sambenz. *URingPaxos*. 2019. URL: https://github.com/sambenz/URingPaxos/.

[30] *Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow*. 2018. URL: https://eng.uber.com/soa/.

[31] *SoundCloud for Developers*. 2018. URL: https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith.

[32] F. R. Tvedt. "Publish Subscriber for Atomic Multicast: A Content Based PubSub System using Kafka". In: UiS. 2018.

[33] F. R. Tvedt. *Github - AtomicKafka*. URL: https://github.com/Destidom/AtomicKafka.

[34] P. Verissimo, L. Rodrigues, and M. Baptista. "Amp: A highly parallel atomic multicast protocol". In: *ACM SIGCOMM Computer Communication Review*. Vol. 19. 4. ACM. 1989, pp. 83–93.

[35] *What are microservices?* 2018. URL: https://microservices.io/index.html.

[36] *What is Apache Kafka?* URL: https://www.confluent.io/what-is-apache-kafka/.

[37] *Who is using microservices?* 2018. URL: https://microservices.io/articles/whoisusingmicroservices.html.