Enhancing Facebook's RocketSpeed Publish/Subscribe System

Author: Dan Danaila

Supervisor: Dr. Peter Pietzuch

Second Marker: Dr. Thomas Heinis

Department of Computing
Imperial College London

Submitted in part fulfilment of the requirements for the degree of
MEng Mathematics and Computer Science

Imperial College London June 2016

Abstract

The growth in the number of people that use social networks and the number of features that they offer requires a scalable and robust publish/subscribe system. RocketSpeed is a new publish/subscribe system developed by Facebook targeted to be used by multiple applications such as Facebook Messenger, GraphQL-cache, WhatsApp, etc.. RocketSpeed developers claim that the system can scale up to a trillion topics. However, the open source implementation lacks this ability as it uses an internally developed log storage. One of the main contributions of the project is to overcome this major drawback of the open-source implementation by extending RocketSpeed to use Apache Kafka for message storage. Thus, the project provides a working solution for real-world deployments. The report will further explain the approach taken to integrate Apache Kafka into RocketSpeed along with an evaluation of the new architecture of the system. The project aims to furthermore enhance the functionality of RocketSpeed by providing a Smart Flow Control mechanism that would automatically regulate the incoming traffic. The goal of this throttling mechanism is to ensure the stability of the system by maintaining a balance between the average serve time and throughput. The report will offer valuable insight of two proposed algorithms that exploit Little's Law in order to asses the current state of the system and adjust the bound on the number of in flight messages. In addition, the strengths and weaknesses of both algorithms are highlighted and discussed to determine the most suitable algorithm for the Smart Flow Control mechanism.

Acknowledgements

Firstly, I would like to express my sincere appreciation to Peter Pietzuch for his guidance, help and invaluable feedback throughout the project. Secondly, I would like to thank my second marker, Thomas Heinis, for his advice and assistance.

Furthermore, I would like to thank Dhruba Borthakur and Peter Alexander from Facebook for providing me invaluable feedback and answering questions about RocketSpeed. Last but not least, I would like to thank my family for their unconditional love and encouragement throughout my time at Imperial College London.

Table of contents

Chapter 1

Introduction

1.1 Motivation

The publish/subscribe paradigm is not a new concept in the software engineering world. However, today it is highly used in a wide range of applications. For example, the Gmail and Yahoo apps use this paradigm to fetch the latest emails. Another app that is used daily by hundreds of millions of people is Facebook. It relies on this pattern to bring latest news to the phones and web browsers of people by letting them subscribe to friends' activity, different public pages and other content.

Facebook is the most popular social network at the moment and has been expanding in the past couple of years on several levels, including the number of active users, and functionality that it provides. The social platform was initially intended for Harvard students, then for the students in the Boston area and since 2006 it is available for anyone in the world. We can observe that since 2008 Facebook has been growing linearly with approximately 200 million users per year, at the end of 2015 exceeding an average of 1.5 billion active users per month, as we can see in Fig 1.1.

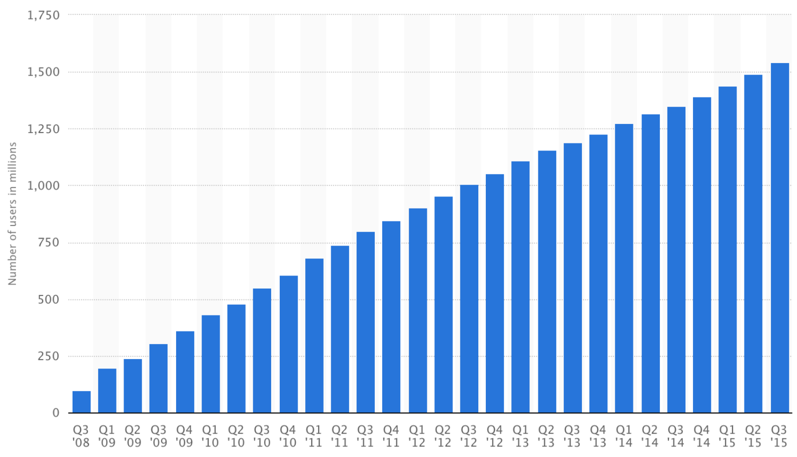


Fig. 1.1 Facebook overall monthly active users [1]

Initially, Facebook's functionality was limited to posting statuses and images. However, it continued to expand by introducing Public Pages that can be owned by different organizations, famous people, etc.. However, the essential part is that this functionality was enhanced by introducing the possibility to subscribe to friends' and Public Pages' activity. In addition, in 2008, Facebook introduced Messenger, an instant messenger that allows users to communicate. We can observe that over the time, most of Facebook's newly developed features have been based on the pub/sub paradigm. Correlating the number of users' growth and the functionality development (which was primarily based on the pub/sub paradigm), we can observe that the number of messages (e.g. posts, images, videos, etc.), producers and consumers has increased dramatically. Moreover, we have to take into account that there has been an even more rapid growth in the number of monthly mobile active users as we can observe in Fig 1.2. Using the mobile app every now and then to check the latest freinds' stories, posting a status or sending a few messages can take less than 30 sec. This will amplify the volume of messages passing through the social network even more, as people can use their phone almost at any time.

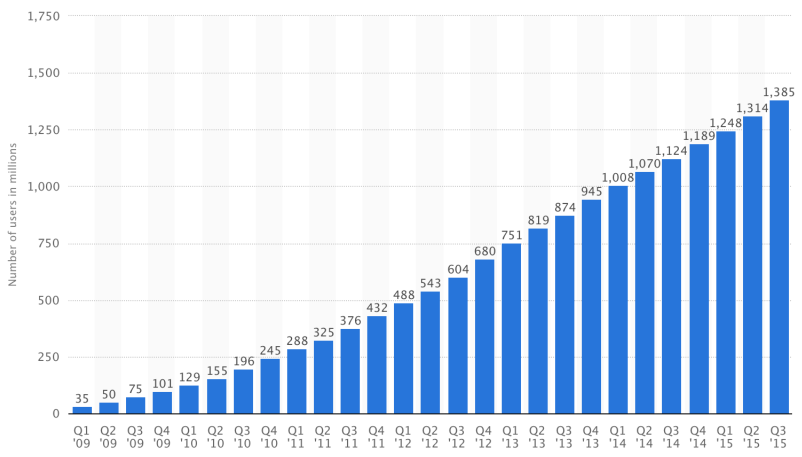


Fig. 1.2 Facebook mobile monthly active users [2]

As a result of the above analysis, there is a clear need of a new publish/subscribe system that can scale up to support such a huge number of producers, consumers and messages. This is why Facebook decided to solve this problem by developing a new publish/subscribe system called RocketSpeed.

1.2 Objectives

Designing and building such an ambitious project from scratch is an extremely difficult and time consuming process that would be beyond the time limit given for a masters project. The main objectives of the project are to investigate the open-source implementation of RocketSpeed and enhance it. The project starts by doing a detailed analysis of the system's architecture and identifies a series of problems. Firstly we aim to improve the scalability of the system and provide a solution for real-world deployments by integrating Apache Kafka as a message storage for RocketSpeed. Secondly, we aim to increase the robustness of the system by adding a Smart Flow Control mechanism that would protect the system from becoming overloaded.

1.3 Contributions

This project makes the following contributions:

- identify a major scalability issue in the open-source implementation
- improve RocketSpeed's scalability by implementing a `C++` extension that allows Apache Kafka to be used for message storage

- a comparison between the performance of the open-source implementation and the one using Apache Kafka
- devise and implement two novel algorithms that based on the analysis of several metrics regulate the incoming number of messages
- a detailed analysis of the two algorithms performance identifying strengths and weakness

Chapter 2

Background

2.1 The Publish/Subscribe Pattern

In software engineering, publish/subscribe pattern is a well-known paradigm for message communication. A system built using this pattern allows Publishers to connect to the system and send messages, which can have different characterizations (e.g. a message can be published to a particular Topic). As a result of this design, the Publishers do not know who the recipients of the message are. The other key element of the publish/subscribe paradigm is the Subscriber, which expresses its interest in a specific set of messages that are published through the system. This set of messages is determined by a characterization established by each Subscriber (e.g. a Subscriber can be interested in two specific Topics).

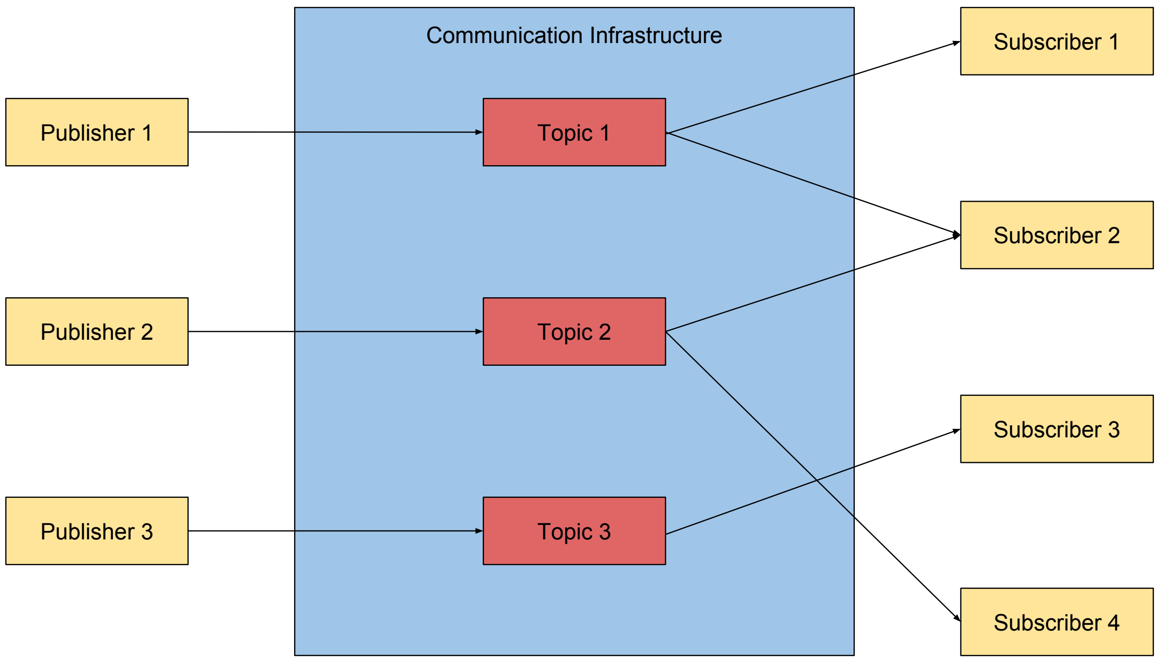


Fig. 2.1 An abstract structure of a pub/sub system

2.1.1 Filtering

As described above, Subscribers need a way to get only the messages that they are interested in. Naturally, there appears the need of some sort of filtering of the messages. However, in order to be able to filter a set of data you need to first characterize the data. In the literature, there are two main methods of message filtering: topic-based and content-based.

Topic-based filtering assumes that Publishers will produce messages that are published under virtual channels that have a specific name. A good example would be Piazza, the website used by Imperial College to provide students with a means to discuss about different courses. In this case the topic would be the name of the course, e.g. Parallel Algorithms, Software Reliability, etc.. On the other hand, Subscribers will be able to express interest in a specific set of topics by just providing the system with a list of topics. This way they receive just the messages of interest and ignore all others. In our example, students would just need to subscribe to the classes they are enrolled in and will get notifications about the relevant courses.

Content-based filtering offers a richer characterization of the messages. It requires messages produced by the Publishers to have several attributes. Now, by defining relations: $=, >, \geq, \leq, <$ on those attributes we can obtain filters that offer a richer expressiveness. To make it even more flexible filters can be combined using logical operators *and* and *or*. To illustrate this, think about the stock markets. Let's say that you own X Google shares and want to sell them at some point, but don't want to constantly watch the stock market. You can subscribe to NASDAQ, an American Stock Exchange, to receive notifications of the current price of Google shares only if the price exceeds a limit, or decreased under a specific point, or some other index goes over a limit, etc.. This way you can describe exactly when to receive the new price so that you can make an informed decision.

2.1.2 Loose Coupling

One important aspect of the system that is worth mentioning is the loose coupling of the components of the system. Decoupling can be noticed on several levels.

The first level of decoupling that probably pops in anyone's mind when thinking about a pub/sub system is **space decoupling**, as regarded by the literature. This assumes that Publishers and Subscribers do not need to know or care about each others' existence. The system is focusing on getting messages from Publishers and routing them to the correct Subscribers. This way, we can notice that if a Publisher goes down, it will not affect the system in any way. It would be the same as not publishing anything. Also, if a Subscriber disconnects, then the system will no longer have to deliver messages to it. Moreover, no Publisher will be affected as they talk only to the system to publish messages and have no knowledge of the subscribers.

Another important decoupling that most of the systems also implement is **time decoupling**. This has some interesting implications. First of all, the ability of the system to deliver messages from Publishers that started publishing after a part of the subscriptions were issued. Moreover, some systems store messages for a period of time, thus, allowing Subscribers to disconnect from the system and receive messages at some later point in time when they rejoin the system.

Last but not least, there is the **flow decoupling**, which takes place at local level. This concept assumes that the publish procedure of the Publisher and message consumption procedure of the Subscriber are non-blocking.

2.2 Related Work

2.2.1 Wormhole

Another publish/subscribe systems developed at Facebook and made public is Wormhole [4]. This system was designed to reliably replicate data across data centers.

Problem definition

The main use of Facebook is to share content of any type: statuses, images, videos, etc.. When the user posts, the event is logged to a database. There are multiple services that are interested in knowing about this event. For example, News Feed which uses this information to notify the user's friends about the post. Another such example is cache invalidation. To illustrate this, think about the cache of user data for which it is crucial to know when the user changes some privacy settings.

Making each application poll the databases that it is interested in would not be feasible. There are two possible cases that can occur. The first one is the application would poll the database at some long time intervals, which would result in stale data and inconsistencies. The other one is the application would poll the database often, which would result in a slow down of the storage system and performance decline.

Furthermore, Facebook stores data in different storage systems as MySQL databases, HDFS and RocksDB.

Specific terms' definitions

- **Dataset:** A set of related data. An example of dataset is data generated by a single user.

- **Shard:** At Facebook there is a huge amount of data flowing in the system. This is why it cannot be stored in a single place. This is why data is usually split in very small chunks, datasets, that are then grouped on different shards. An example of shard can be the group of data generated by users who were born in a town. However, usually shards contain approximately the same amount of information.

- **Datastore:** A datastore is a collection of shards.

- **Publisher:** Each datastore runs a publisher, which reads updates from the log of the storage system and generates notifications in a standard key-value format, referred to **Wormhole update**.

- **Subscriber:** Applications that use Wormhole are divided in different shards. The set of shards on the publisher site does not need to match the set of shards on the subscriber, and it is highly likely that the shards on a producer are distributed to different subscribers.

- **Flow:** The stream of updates for a shard.

- **Datamarker:** Metadata per flow indicating the latest update processed by the subscriber. It is updated after the subscriber acknowledges the updates.
- **Caravan:** A caravan is a reader that retrieves updates from a specific point in time and publishes them for all the flows that are on that particular datastore (i.e. publishes updates for all the shards on that particular datastore).
- **Lead caravan:** The lead caravan is the caravan that reads and publishes the latest updates.

Wormhole's workflow

Caravans are one of the key elements of how Wormhole works. The most important reason why flows are grouped under caravans is to reduce disk I/O. Under normal conditions there should be only the lead caravan that reads and publishes updates for all the flows. However, some of the flows might fall behind or failures might occur and flows might diverge. In such cases new caravans are created and flows are assigned dynamically to the nearest caravan in order to optimize I/O costs. However, obviously this means that flows that are farther ahead in time inside a caravan would have to wait for the other flows to catch up. Making flows wait for other flows is not desirable. However, if there would be allowed a larger number of caravans, then larger I/O costs would be incurred. So there has to be found a balance between the number of caravans and the grouping of flows' datamarkers. Other interesting operations are caravans split and caravans join. When the flows inside a caravan are too farther apart the caravan would split. Also, if the flows in two different caravans are close then the caravans would join.

Furthermore, message filtering is done at publisher level. To make use of this feature, the subscriber application can inform the publisher about the filters to apply to updates. This optimization reduces network traffic by not transmitting useless data. Taking into account that these applications can run in different clusters in different geo-locations, saving traffic network can decrease latency and allow the network to transport more useful data.

Reliability is particularly important for Wormhole. Just consider the case when one update is missed. It can lead to data corruption or leave the data in an inconsistent state. This is why Wormhole offers Single-Copy Reliable Delivery (SCRD) and Multiple-Copy Reliable Delivery (MCRD).

SCRD guarantees that when an application is subscribed to a single copy of a dataset it will receive updates at least once. Moreover, it is important to note that Wormhole guarantees ordered updates of a particular flow. In order to achieve this, Wormhole uses the reliability of the subscriber's TCP connection to the publisher combined with datamarkers. In the testing phase, it was discovered that if it relied on the application layer acknowledgments for every update then it would result in a heavy bandwidth utilization and low throughput. To improve this, datamarkers were introduced, which are send periodically by publishers to subscribers as checkpoints. In case of network or subscriber failure, when the connection is reestablished the publisher knows from what point to restart each flow because it also stores the datamarker for each flow.

MCRD guarantees that when an application is subscribed to multiple copies of a dataset it will receive updates at least once. Basically, MCRD works by using SCRD and in the case when a publisher fails, then it connects to another publisher. However, there is a catch. In SCRD we explained that datamarkers are stored on the publisher site, but if the publisher fails then they are lost. To overcome this, MCRD uses Zookeeper, a highly available distributed system, where datamarkers are published. However, after solving this, there appears another issue. The datamarkers are pointers specific to the logs on the publisher site, so just exporting those datamarkers would not be too helpful. To overcome this, logical positions were introduced. These have the property that they indicate to the same place in both files and knowing the logical position you can compute the exact point in the log.

Wormhole, as described by Facebook's research team [5], is a huge success and manages to achieve its goal without problems. To emphasize this they claim that just for the UDB deployment, Wormhole handles over 1 trillion messages a day. Also, it reduced CPU utilization on UDBs by 40% and I/O utilization by 60%. Furthermore, it improved cache consistency across data centers.

To summarise, Wormhole is a highly reliable publish/subscribe system that is focused on publishing updates that take place at database level. We consider it to be a highly efficient solution targeted at taking load off the databases, which achieves its goal. Moreover, the explanations about the caravans are very useful as they present exactly the trade-offs between having many readers (incurring large I/O costs) compared to having a smaller number of caravans (incurring delays for the flows that are ahead). We can learn from this, as the Control Towers in RocketSpeed use tailers to read data from the Log Storage. The tailers have the exact same role as the caravans have for Wormhole, hence, as indicated by the results section, an optimal number of such caravans is two. Thus, we can use this indication, to get a balance between latency and I/O costs.

2.2.2 Thialfi

Thialfi [7] is a publish/subscribe system designed and developed at Google. Its' main goal is to ensure data freshness for applications that run on a cloud infrastructure environment.

Problem definition

There are many applications that use shared data between multiple users, multiple devices and the cloud infrastructure. Clients usually maintain a cache of the data and it is important that this cache is maintained up to date. For example, multiple users might attend a common meeting. Then if someone changes the time of the meeting is critical that everyone receives the update. A large number of such applications were developed at Google. For instance, Google Drive let users share documents and edit together in realtime, Google Calendar allows synchronization of events between multiple users, Gmail has to synchronize latest mails on multiple devices, etc.. As there so many applications that rely on shared objects it makes sense to have a common notification system that would reliably inform all interested parties of changes.

Goals

1. **Tracking:** System is responsible for storing a map from objects to interested parties.
2. **Reliability:** Notifications should be distributed reliably and developers should not have to worry about failures.
3. **End-to-end:** System should provide end-to-end service despite use of unreliable channels.
4. **Flexibility:** System should support applications written in multiple languages.

Design

The first important design decision made in Thialfi is to represent shared objects as versioned objects. The service just propagates the latest version number for all the objects, not the actual data of the object. One reason to do this is the fact that different applications use different formats of data, so this way it would remain general enough to work for all applications. Furthermore, clients may be offline for a long period of time, which can lead to long queues of updates. These will occupy a lot of server resources, even waste them in case the client never becomes online.

Another important, but quite natural assumption made by Thialfi is that the version numbers are increasing numbers so that it can easily identify the latest update. For synchronous stores this is achieved by incrementing the version number after every update. However, for asynchronous stores this is more difficult. To avoid modifying asynchronous data stores, developers decided that it is enough that when an update hits a replica to publish to Thialfi the replica's current timestamp. They make the assumption that clock skew inside data centers is very low so the probability of having clashes or missing updates is low.

Thialfi is divided, as expected, in a client library and server site service.

The client library allows the application to subscribe to shared objects. When there is an update, the library has a callback that informs the application of the new version of the object. After that, it is the application's responsibility to talk to the server and get the latest version of the object, as Thialfi does not provide any means to talk to any kind of storage, as presented above.

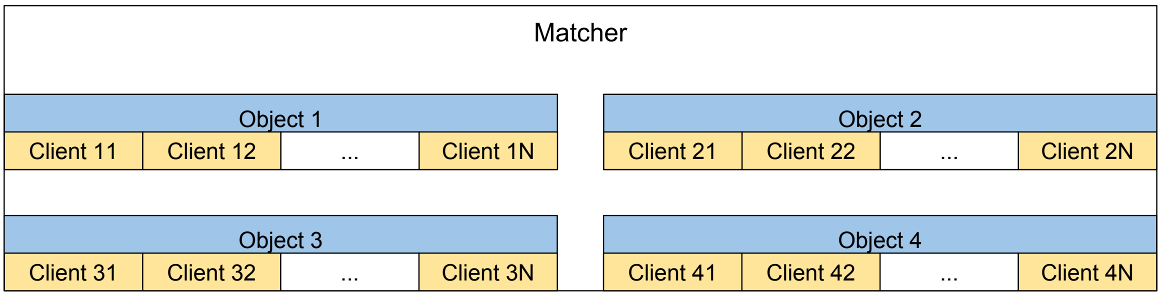


Fig. 2.2 Matcher's architecture

In the data center there can be found two types of servers: **Matchers** and **Registrars**. Matchers are divided by objects and receive and propagate notifications. They store the latest version for an object and also a list of interested clients, as it can be observed in Fig 2.2. Registrars are divided by clients and manage communication with the client. They store for each client, a list of objects that it is interested in, as it can be seen in Fig 2.3. So there is a multiple view of the state used by Thialfi. This way data can be accessed either by using the object id or client id, and the other state is update asynchronously. Because the updates between the servers are done asynchronously, both servers also have a list of pending updates send to the other one.

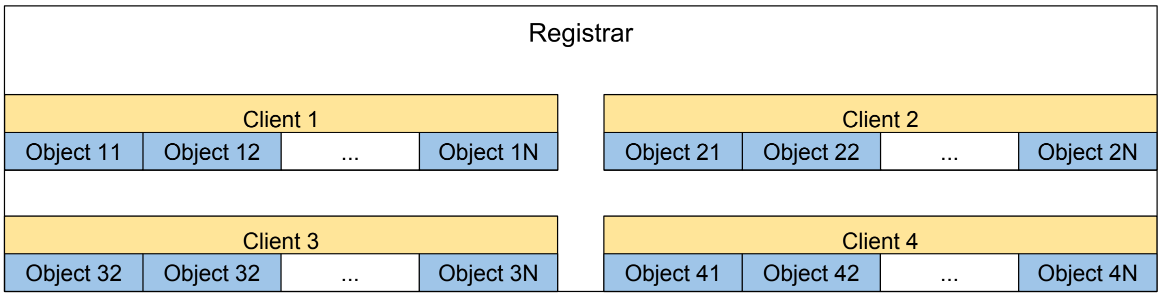


Fig. 2.3 Registrar's architecture

Moreover, both Matchers and Registrars use an in memory design. So they do not provide data persistency. To solve this, Thialfi uses Bigatble, a distributed storage system designed and developed at Google. Whenever, Matchers and Registrars do an operation they register it in Bigtable. This is used for a faster recovery in case of failure.

Evaluation

Thialfi has been in production at Google since the summer of 2010. According to the claims of the developers it scaled linearly to millions of users and latencies have remained stable. We can conclude that Thialfi is a system with highly limited functionality, but it achieves its goal. It successfully manages to keep applications informed about the latest updates without wasting resources and eliminates the need for periodic polling by storing the interested clients in memory. Thialfi can be viewed more as a notification system that just indicates that new updates are available and passes the responsibility of getting the updates to each application.

2.2.3 BlueDove

BlueDove [6] is an attribute publish/subscribe system focused on scalability and elasticity. It manages to achieve its target by organizing servers into a scalable overlay and by exploiting skewness of data.

Problem definition

In the recent years we can identify several trends in the computing world. One of them is the high increase in the number of applications that are sensor-based, i.e. change their response by continuously reading several real world sensors. For example, an app that gets latest updates for transport in London, i.e. tube statuses, buses approximate position on the map, etc. can offer users different routes based on traffic conditions, tube delays, etc..

Another popular trend is cloud computing, also known as ”on demand computing”. It has faced huge success in the past years as it eliminates the cost of investing in expensive infrastructure and allows users and businesses to use exactly as much resources as needed and pay just for what they use. A good example, would be a small business that does not afford to invest in infrastructure to support its product. However, initially their product will not be very popular, so they would not use too much resources, so by using cloud computing costs will be low. This would allow them to invest capital in publicity, for example, thus making their product more popular. Another example that helps illustrate the usefulness of cloud computing is represented by researchers that might want to run different experiments, but do not have sufficient funding to buy all the necessary equipment. This way they can use cloud computing for running several experiments and pay just for the period of time that they are using the resources. This can be very cost efficient, as it allows a much smaller investment for testing a larger potential solutions for different problems.

Challenges

Combining the two trends we obtain an interesting problem. How can we build a cost efficient application relying on sensors updates (possibly millions of updates/second) and that can potentially be used by tens of thousands of people at the same time? Such a problem poses interesting challenges. First, the system has to be scalable to sustain a high number of updates and high number of users. Additionally, it has to be elastic, to adapt to current workload of the system. At peak times, when the number of updates and/or the number of users grow, the system has to be able to request multiple workers and divide the workload among them. Furthermore, when the system becomes underused, i.e. number of messages and/or number of worker diminishes, the system should be able to detect this and free some of the workers that it uses and rebalance the work among remaining workers.

Solution

First of all, BlueDove is an attribute based pub/sub system. This means that each message is characterized by k attributes. Let V^i be the space for attribute i and $V = V^1 \times V^2 \times ... \times V^k$ be the attribute space. Also, in attribute-based filtering the operations: $=, >, \geq, <, \leq$ are defined for each attribute. So the filters that are applied over an attribute are usually unions of intervals. To illustrate this better, consider that we have as attributes the x and y coordinates in the plane and we can issue an filter over the x coordinate: $0 \leq x \leq 10$ or $15 \leq x \leq 20$. This corresponds to the union $[0, 10] \cup [15, 20]$ over the x coordinate.

Furthermore, network topology is a very important aspect of a pub/sub system. Usually pub/sub systems use a multi-hop overlay network. The reasons to employ such a network overlay are unreliable links and high node churn rate due to possible failures/joining of nodes. In a datacenter these problems should not appear as much as the links are more stable and the node membership is more stable as well. In such case a multi-hop network would only increase the time of delivery. Thus adding or removing nodes are not complicated operations.

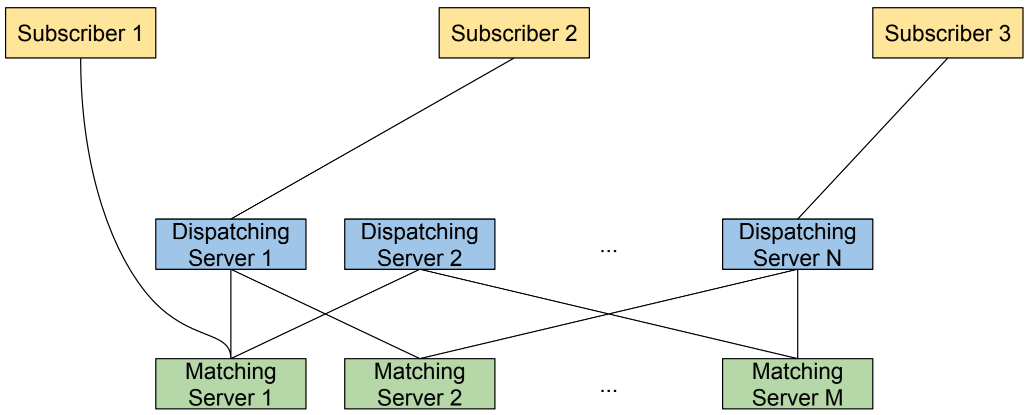


Fig. 2.4 Bluedove's topology

BlueDove utilizes a two-tier architecture, as it can be seen in Fig 2.4. It uses one set of nodes, called **Dispatchers**, as proxies for subscribers and publishers to connect to the system and publish/request messages. The second layer of servers, called **Matchers**, represent the back-end of the system. They are responsible for storing latest updates and then routing them directly to subscribers or by returning a handle to some temporary storage that the subscribers can poll. The second delivery method is mostly useful for subscribers which may not be able to listen on an IP/port for messages. This design has the advantages that messages traverse only one hop and allows grouping similar subscription.

In the context of attribute-based filtering similar subscriptions can be regarded as queries over closed range intervals. This is one of the key optimizations that BlueDove applies. Now, let's see how it partitions subscriptions to matchers. First, let's assume that there are N matchers. Then each interval described above is divided in N disjoint intervals: V_1^i, V_2^i, ..., V_N^i. Then, each Matcher will be responsible for one of the above intervals, for each attribute. This means that each subscription will be assigned k times to Matchers. Moreover, for a particular dimension, there might be multiple Matchers responsible for serving it, if the predicate intersects more intervals.

In order to know where to route message and where to send subscriptions, Dispatchers maintain a global view of how intervals are split among Matchers. However, unlike one might think the messages are send to only one Matcher. For choosing that Matcher there had been studied two policies: subscription based and adaptive policy.

Subscription based policy chooses to send a message to the candidate Matcher that has the least number of subscriptions, as shown in Fig 2.5 . However, this does not take into account how popular a subscription might be. If we take into account the queued messages on a Matcher, then we might end up with a Matcher that has a relatively low number of subscriptions, but a high load. This implies that delivery time will become high.

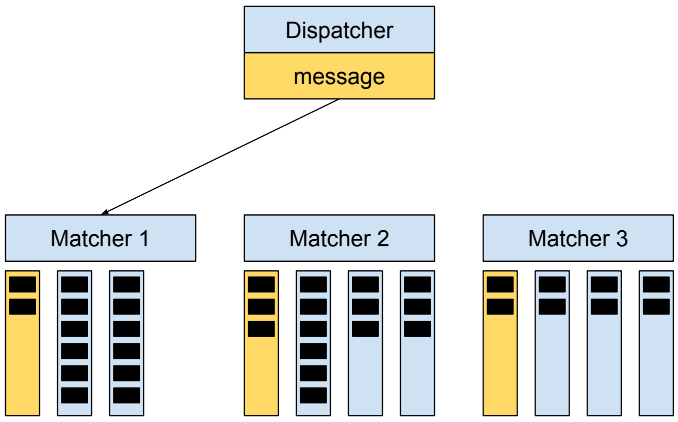


Fig. 2.5 Subscription based policy

On the other hand, **adaptive policy** chooses the candidate Matcher that has the shortest processing time. Processing time is computed for a Matcher by estimating the total number of queued messages, incoming rate and processing rate. This has given better results than the first technique, as it takes into account the whole load of the Matcher. By comparing Fig 2.5 and Fig 2.6 we can observe that **subscription based policy** chooses Matcher 1 because it has 3 subscriptions, while **adaptive policy** chooses Matcher 3 with 4 queues, but far less messages. Hence, **adaptive policy** will chose the Matcher with a lower serve time.

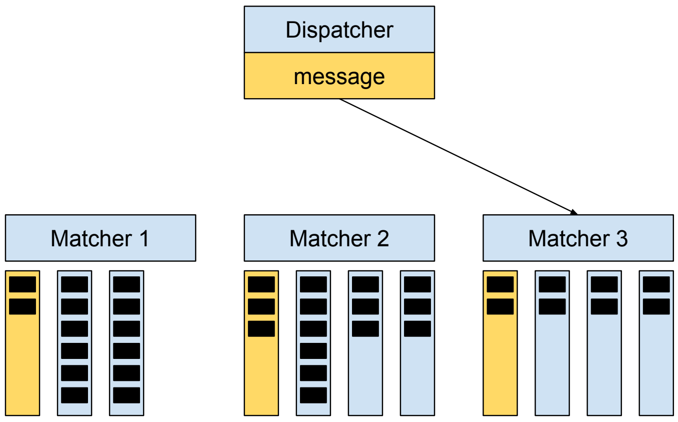


Fig. 2.6 Adaptive policy

Elasticity

One of the primary goals of BlueDove is to provide elasticity. This is why it allows introduction of new Matchers when the system is saturated and elimination of Matchers when the system is underused.

When a Dispatcher detects that the overall workload on the Matchers increases beyond a limit, it introduces a new Matcher. This new one detects the one with the highest number of subscriptions, it divides each interval that Matcher was processing in two intervals, picks one and also takes some of the subscriptions. When a Matcher leaves the Matcher with lowest number of subscriptions is chosen. Then it gets the subscriptions of the old Matcher and also, the intervals for each attribute are joined.

One weakness of BlueDove is the fact that it allows loss of messages. If a Matcher fails, Dispatchers might still send updates to it until they realize it crashed. This means that those messages are going to be lost. The number of messages that can be lost is quite low as the Dispatchers check regularly Matchers, but it is important to note that this can happen.

Evaluation

Analyzing the results of the experiments ran on BlueDove, we can observe that it behaves quite well. The first test run on BlueDove was to fix the number of subscribers and the number of matchers and check what is the maximum rate supported by the system. After using more than 5 Matchers, the results became impressive. With 40,000 subscribers, each extra 5 Matchers increased the maximum message rate with approximately 40,000 messages/sec, i.e. 1 message/sec/subscriber. Now, by fixing the number of messages to 100,000 messages /sec and varying the number of subscribers similar results were obtained. Approximately, every 5 Matchers added support to 15,000 subscribers. The numbers from the two experiments are similar, as both of them show that 20 matchers can sustain approximately 40,000 subscribers at a rate of approximately 100,000 messages/sec. However, these results are not very precise as the skewness of data can influence the behavior of the system.

Overall we consider BlueDove to be a good pub/sub system for domains where data can be partitioned based on attributes and message loss is accepted. However, another guarantee that the system does not manage to make is message ordering. As explained above, BlueDove proposes two algorithms of storing messages on different Matchers, based on the current state of the Matcher. This implies that once a Matcher becomes overloaded messages in a specific interval will be transferred to another Matcher. Thus, no message ordering can be assumed. While the system proves to automatically scale up and down according to the incoming traffic and data skewness, message loss and the lack of any message ordering are two major drawbacks that suggest that the system is not mature yet.

2.3 RocketSpeed

2.3.1 Motivation

RocketSpeed is a distributed publish/subscribe system that aims to be a reliable and low latency delivery pipeline. Initially, its' main goal was to reliably deliver data from a cloud service to mobile devices and the other way around with low-latency. However, as a result of the system design, RocketSpeed can be used to transfer data between different software components geolocated, or between cloud services that run in different geographical locations, or mobile devices and cloud services.

RocketSpeed is a topic-based pub/sub system and it makes the strong guarantee that it delivers messages in order for each topic. However, it doesn't guarantee that it delivers messages in order for different topics. Using this strong assumption a wide range of distributed systems can be built. For instance, it can be used for distributing event notifications, e.g. if someone sends you a message on Facebook, you can receive it on multiple devices that are connected to the internet and run the app. Furthermore, it can be used to refresh distributed caches. To illustrate this, think about the case when one changes privacy settings on Facebook using the browser. It is crucial that all devices that run Facebook should receive the latest updates, otherwise, those devices can leak sensitive information. Other usages can also include distributing a large number of tasks to multiple workers inside a cluster. Furthermore, it can be used by a configuration management system of a distributed service to update configs for the registered components of the service.

As we can observe RocketSpeed has similar functionality as other existing server technologies as Scribe [3] and Wormhole [4]. However, the design and goals of these two systems are different from RocketSpeed's. They are mainly used for transferring bulk data and so, their main aim is to achieve a large through-put of data. Another key difference is that they can scale up to one million topics, while RocketSpeed's goal is to scale up to one trillion topics.

2.3.2 Goals

1. **Multiple readers and writers:** Multiple producers are allowed to publish messages to the same topic and also, different consumers can subscribe to a topic.

2. **Ordering of messages:** Messages produced on a specific topic by multiple publishers are delivered in-order. This means that all subscribers will receive the messages in the exact same order. Note that this does not imply ordering among different topics.

3. **Reliability of message delivery:** A message is guaranteed to be delivered at least once to the consumer. The system will try to minimize the cases when a message is delivered more than once to a consumer.

4. **Number of topics:** One cluster should scale to serve up to one trillion topics.

5. **Priority delivery:** Messages can have two priority types: normal and high. Most of the messages will have normal priority. High priority is reserved for messages that need to be delivered as soon as possible, so they do not have to respect time ordering in respect to the normal priority messages.

6. **Retention of messages:** In order to achieve time decoupling the system offers the possibility to save the message in the system for 1 hour, 1 day or 1 month. After the retention period expires, the messages are purged from the system.

7. **Message Expiry:** A message can also have an explicit expiry period. This does not imply that the message is purged from the system after the expiry period elapses, it just guarantees that the message will no longer be delivered to any consumer.

8. **Selective Delivery:** A subscriber might not be interested on all the messages published to a particular topic. So, it can define a list of attributes. On the other hand, a producer can decide to publish some messages only to a subset of attributes, thus not polluting all the subscribers with unwanted messages. This feature tries to combine topic-based filtering and content-based filtering, so that the system can be more expressive.

9. **Low Latency:** Under normal conditions, RocketSpeed aims to deliver messages with low latency.

10. **Failure Report:** In case RocketSpeed fails to deliver a message it will notify the subscribers and will also provide a reason for the failure.

2.3.3 Design

The first design decision to be noted is that the API provided by RocketSpeed to publishers and subscribers is encapsulated as a SDK for iOS and Android platforms and in a library available for cloud services. The reason to include both components in the same SDK and library is the fact that in many use cases the same client will be a publisher, but also a subscriber. Take as an example the Facebook app, which lets you subscribe to friends' activity, but also lets you produce stories by posting text, images, etc.. So it would make sense to have both components available.

To have a scalable delivery pipeline, the code for receiving and routing messages will be running in the cloud. The API provided in the SDK for mobile platforms and the library will be the glue between publishers, subscribers and the delivery pipeline.

The delivery pipeline that runs in the cloud consists of multiple components that have to handle message receival, storing the messages for each topic based on the retention period and routing to the messages to the correct subscribers. To be able to achieve this functionality the system has been split into multiple components that ensure scalability of the system and a nice flow of the information through the system.

2.3.4 Data Delivery Pipeline Components

Pilot

The Pilot is the software component that is responsible for handling the messages coming from the producers, putting them to the corresponding topic, ensuring that the message has been stored properly in the system and then informing the Producer if the above operations were successful or not. The steps taken by a message to be published are presented in Fig 2.7. In order to ensure that the messages are persistent in the system, the Pilot stores them to Log Device. To map a particular topic to a log, the Pilot uses consistent hashing on the topic name. This is a very important observation, as this idea eliminates the need to store a routing table from topics to logs. Thus there is no bound introduced on the number of topics that the Pilot can write to as it uses the hash function to determine the log.

Another essential question that has to be answered is: how does a producer connect to a Pilot? There are two cases that have to be analyzed. The first one is when the producer is co-located with the Pilot. In this case the producer will connect to that producer as it will eliminate one hop. In the other case, the producer will connect to a random Pilot and publish messages. In case the load on Pilots increases the solution is to add more Pilots, as the publishers do not differentiate them.

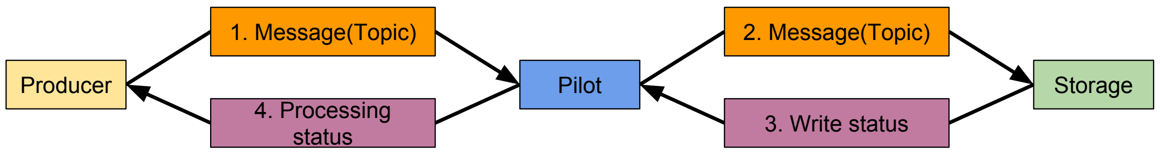


Fig. 2.7 Steps taken for message publishing

Log Storage

As RocketSpeed wants to ensure data retention for three distinct interval of times it has to store the messages somewhere. To be able to achieve message persistency, RocketSpeed needs a storage system that can provide read/write functionality. One advantage of exposing just this functionality through an API is the fact that it makes the storage pluggable. In addition to this functionality, this storage system needs to serialize writes, so that any reader would get the messages in the same order on a particular instance of Log Storage.

For this part of the project Log Device was chosen to be used. It is a storage system optimized for storing logs and that can support up to a million logs. This is the reason why multiple Topics have to be hashed to the same log, as described in the Pilot section. Another advantage of Log Device is that it streams new entries in the logs to readers.

However, for the project we will not be able to use Log Device as it is a system developed internally at Facebook. So for the scope of this project we will be using a mock implementation of the Log Storage.

Copilot

Another crucial question to be answered is: how do consumers get the messages from RocketSpeed? As described above, the messages are stored in Log Storage. However, it would not be feasible for the consumers to connect directly to the Log Storage for multiple reasons. First of all, a consumer might be subscribed to Topics that reside in different logs that are stored on different machines. Furthermore, as we expect the number of subscribers to be large, one server that would handle just the storage of logs would not be able to cope with the throughput of the requested information.

The software component that allows subscribers to connect to RocketSpeed is the Copilot. It is responsible for gathering all the messages that are requested by subscribers. It can be viewed as a proxy for the consumers that subscribe to different topics. Also, consumers will connect to Copilots using the same logic as the publishers connect to Pilots. In case the consumer and the Copilot are co-located, the consumer will connect to that particular Copilot. Otherwise, will connect to a random Copilot. Also, in the eventuality when the connection fails, the consumer will just connect randomly to another Copilot.

Control Tower

As described in the Copilot section, a machine might not store all the topics a particular subscriber is interested in. So it means that the copilots might need to connect to multiple instances of the log storage. Also, there might be more than a copilot interested in a specific topic. This means that multiple copilots can end up connecting to the same Log Storage. However, for the Log Storage write availability, scalability and latency are the most important. Read availability and latency come after these. So we do not want to burden the Log Storage with continuous requests for data.

As described in the Log Storage section, Log Device is optimized for streaming data to readers. In order to take advantage of this we need a software component that would do the reading part and give data to Control Towers. This is exactly what the Tailers are responsible for. They connect to different logs, get data, demultiplex it into different topics and stream it into Control Towers. Also, the Tailers cache locally data. This is used for the subscribers that are falling behind (usually the mobile subscribers). This cache reduces the requests for old data from Log Storage, so helps in reducing the number of random seeks in the Log Storage.

While Tailers get the stream of data from the Log Storage, demultiplex it and then cache it, all this data needs to be routed to the Copilots that require it. This is the reason Control Towers were introduced. Their main aim is to route data to the Copilots by querying the cache of the tailers. In order to get a better flow of the data to subscribers the numbers of Control Towers that tail one log is two. This way Copilots have two different paths to get to a log. However, this number can change, as it is configurable. A Copilot uses a consistent hashing function on the topic name to discover the Control Tower responsible for that topic. This is perfect as it does not consume routing information in the memory of the Copilots. However, the big issue that appears now is that in order for a Control Tower to send a stream of updates on a particular topic to a Copilot, it needs to know how to reach that Copilot. This implies that at the level of Control Towers we need to store a routing table. As any Copilot can connect to any Control Tower, we need to know the address of every Copilot and whether it is subscribed to a topic, or not.

Let's make a simple analysis to estimate how big the routing table can get.

Number of active topics	$p * 10$ billion (i.e. $p\%$ of the targeted number of topics are active)
Number of Copilots	1000
Routing table metadata	$p * 1.25TB$ (p * 10 billion * 1000 bits)
Topic name length	100 bytes
Topic name space in total	pTB (100 bytes * p * 10 billion)
Total routing table space	$p * 2.25TB$

Table 2.1

Based on the analysis in Table 2.1 let's estimate the total number of machines required for Control Towers.

p	Routing space	RAM/machine available	Number of machines
1	$2.25TB$	$20GB$	113
1	$2.25TB$	$30GB$	75
2	$4.5TB$	$20GB$	250
2	$4.5TB$	$30GB$	150

Table 2.2

In the analysis in Table 2.2 we assumed that the machines have $20GB$ or $30GB$ of memory available. The reason for not considering more memory available is the fact that in order to make full use of the cache of the Tailers we will assume that the Tailers run on the same machines as the Control Towers. So it will make sense to try and cache as much data as possible in memory. Let's assume an approximate traffic of 1 trillion messages per day, with an average of 1KB per message. Then we would have an approximate 11.6GB/sec traffic. Thus, let's also observe using the number of machines from the above table how much traffic would we be able to store in memory assuming that we allocate 100GB per machine for routing table and the cache.

Number of machines	RAM/machine available	Total time
113	$80GB$	13 min
75	$70GB$	7.5 min
250	$80GB$	28 min
150	$70GB$	15 min

Table 2.3

Correlating the numbers in Table 2.2 and 2.3, we can observe that an optimal solution that would not require a very large number of machines would be to use 113 machines, allocate $20GB$ for the metadata and $80GB$ for the cache. Thus we will obtain a cache of the last 13 mins.

A simplified view of RocketSpeed's workflow

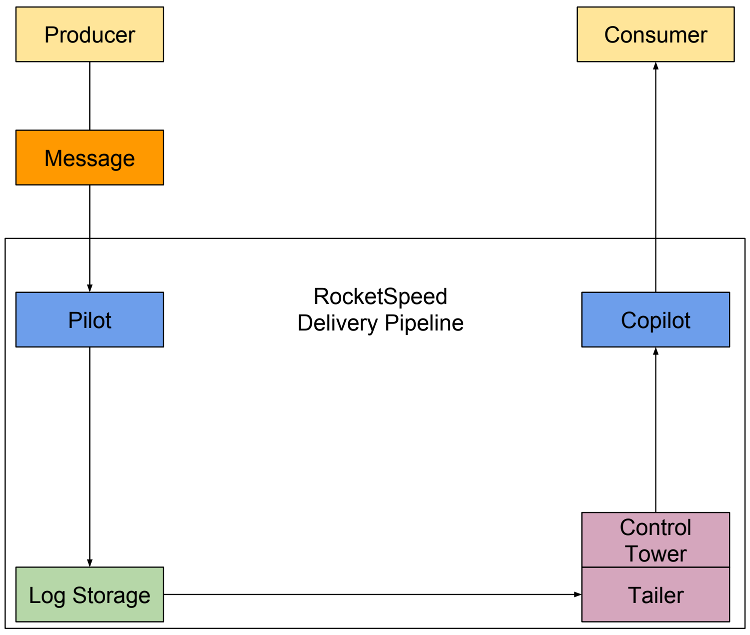


Fig. 2.8 A simplified version of the system exemplifying the flow of a message

2.3.5 Namespaces

Moreover, RocketSpeed allows grouping of topics in namespaces. Each topic will belong to a particular namespace and they have to be unique just inside that specific namespace. As described in the Goals section, RocketSpeed aims to provide retention of messages. This can be set up only at namespace level, they cannot be associated to particular topics.

2.3.6 Multi-Tenant System

Another awesome feature of RocketSpeed is the fact that it aims to be a Multi-Tenant system. This means that only one instance of RocketSpeed will be able to sustain the activity of multiple applications. For example, Facebook Messenger, WhatsApp messages, Facebook and Instagram notifications can be four different systems that can work at the same time on a single instance of RocketSpeed.

This is a really useful generalization, as this means that different services can use the system over time, with some being launched and other being taken down. However, there is an important thing that has to be noticed. This different systems can use different amounts of resources and one service should not degrade the performance of another one. This is why when a new tenant registers to use the system, it will also register a SLA [12] (Service

Level Agreement) with RocketSpeed. The Service Level Agreement of each tenant will specify a maximum input data rate and a maximum output data rate. If a tenant exceeds the input/output data rate, then its' Producers' messages will be rejected and the Producers will be informed. A well behaving Producer might use this as information to induce an artificial delay to its messages so that the rate of messages for the tenant will go down and its' messages will not be rejected.

Finally, RocketSpeed is designed such that it supports a small number of tenants, in the order of thousands.

2.4 Project Initial Set Up

2.4.1 Cloud Computing

The first choice that one has to make when testing and developing a distributed system such as a publish/subscribe system is what cloud computing environment to use.

One of the first considered options was to use Amazon Elastic Compute Cloud (Amazon EC2 [9]). The main benefit of Amazon EC2 is the possibility to choose the number of instances to use, configure security, networking and manage storage. Even though Amazon offers different pricing schemes, from paying for resources utilized by hour to reserving a number of instances for a longer period of time, they all involve payment.

The other cloud computing environment considered was DoC's Private Cloud. This is a private cloud computing environment build by DoC on top of Apache Cloudstack. It proved to be a better solution as it gives a simple interface that allows easy creation of VMs using predefined templates, flexible computing offering and disk offering. Moreover, it has two big advantages, it is free and runs inside Imperial, so we can get assistance from CSG. As for development of RocketSpeed we need just a bunch of computers to run the components of the system and a few computers to run the publishers and consumers, the above properties of Apache Cloudstack make it the best fit for this project.

The first problem encountered when using DoC's Private Cloud was the fact that the web interface is accessible just within the college network. As this would impose a big restriction to work only from college, ssh tunneling was chosen to solve this obstacle. However, the big issue was the command suggested on the website did not work. It was suggested to use ssh with Local Port Forwarding. In order to tackle the issue we ran ssh in verbose mode and the debugging output indicated that the request was forwarded to the correct website and port, but it was not working. After doing research on the internet, we decided to overcome this by using Dynamic Port Forward. This required to set up the laptop to send all the traffic using

SOCKS proxy to Imperial's network. However, this was only required just when the web interface was needed, i.e. for creating and setting up machines.

2.4.2 VMs Decisions

OS

When creating a new VM the first choice one has to make is decide which OS to use. DoC's Private Cloud offers several templates:

- Ubuntu v14.04 30Gb (Non CSG)
- Ubuntu v14.04 50Gb (Non CSG)
- Ubuntu v15.04 30Gb (Non CSG)
- Ubuntu v15.10 30Gb (Non CSG)
- CSG Maintained - Ubuntu v14.04 64bit
- GITRUNNER-CSG

Out of this list we decided to try just the Non CSG images as we did not want to depend on CSG in case of lack of privileges to install any packages that might be needed, or other problems that would arise. The project would not need any special security and the number of packages required for it to compile and run is quite low, so a clean installation of Ubuntu was the perfect choice.

We tested Ubuntu v15.04 and Ubuntu v15.10, but the C++ compiler that gets installed when running the command: `sudo apt-get install g++` is `gcc 4.9`, which, compared to `gcc 4.8` and `gcc 5.2`, gave errors which prevented the code from compiling. Of course there was the option to modify the code to fix the errors, but one of the initial decisions was not to change the code from Facebook Experimental's Github repository and moreover, these errors should not have appeared as the other two compilers did not indicate them. So, instead of trying to manually install one of the latter two compilers we decided to go with the more stable Ubuntu v14.04 LTS which gets `gcc 4.8`.

As there are two versions of Ubuntu 14.04 another choice had to be made. For testing purposes RocketSpeed does not need any space for storage as we are not interested in message persistence, it holds the messages in memory. So, the natural choice was to go with Ubuntu 14.04 30Gb (Non CSG) as the project compiled does not occupy more than $2.5Gb$.

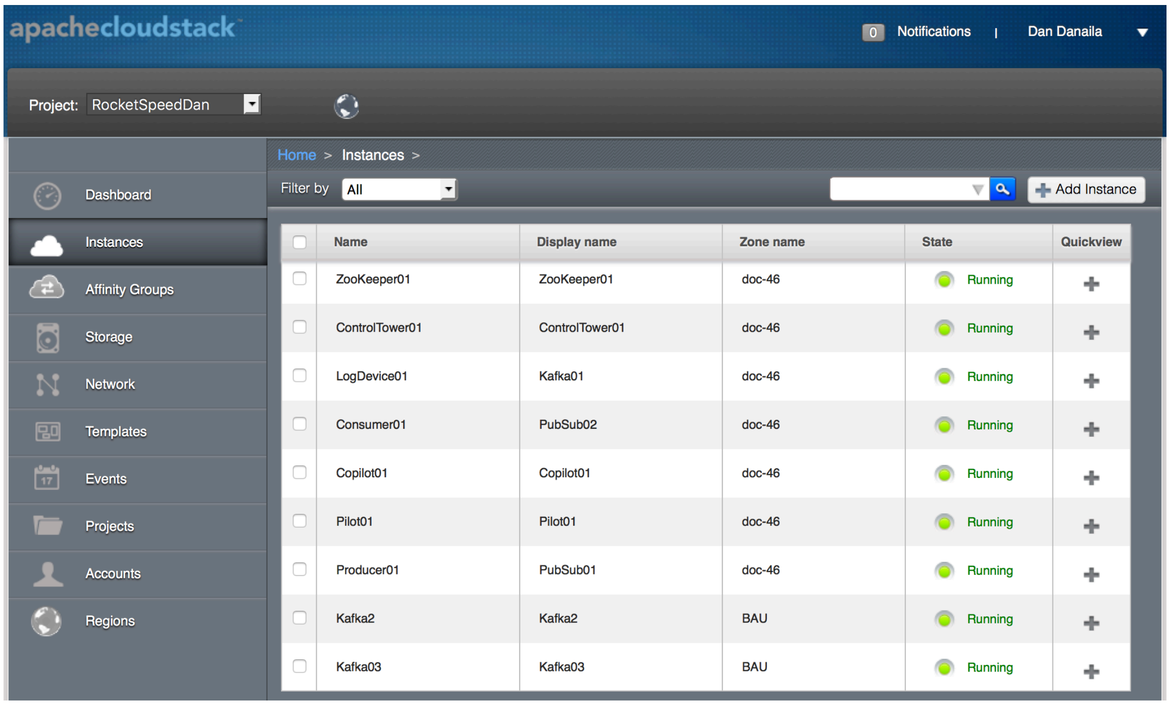
Compute Offering

The next step in setting up the VMs is determining the **Compute Offering**. For each machine we settled to use local storage, rather than shared storage. The main advantage of using local storage is gaining better performance. However, to get better performance we trade in the High Availability property that shared storage has. This is not a big loss, as the system would not benefit from this property either way, as it is in the testing and development phase when it would not run for a long period of time.

For the actual specifications of the VMs I decided that each component of the system should run on identical machines with the following specifications: `CPU: 4 cores, 2.0 GHz & RAM: 8GB`.

Disk Offering

For **Disk Offering** the decision was straight-forward. As explained above, RocketSpeed does not need any additional storage as after the system is shut down message persistence is not required. So, no additional disk storage was requested at this step.



Name	Display name	Zone name	State	Quickview
ZooKeeper01	ZooKeeper01	doc-46	Running	+
ControlTower01	ControlTower01	doc-46	Running	+
LogDevice01	Kafka01	doc-46	Running	+
Consumer01	PubSub02	doc-46	Running	+
Copilot01	Copilot01	doc-46	Running	+
Pilot01	Pilot01	doc-46	Running	+
Producer01	PubSub01	doc-46	Running	+
Kafka2	Kafka2	BAU	Running	+
Kafka03	Kafka03	BAU	Running	+

Fig. 2.9 Initial dashboard of the project

Additional packages

As a cleaner OS tends to behave better and break less than one which has a tone of unnecessary software we settled down to install just the essential packages that RocketSpeed required to compile and a bunch of useful programs for developing. The list includes:

- `git, tig` - primary tools to work with Git repository
- `libgflags-doc, libgflags-dev, libgflags2` - `gflags` library
- `make, cmake` - tools for automating code compilation and installing different libraries
- `g++` - C++ compiler
- `openjdk-7-jdk` - `Java 7` development kit
- `gdb` - primary tool for code debugging
- `valgrind` - primary tool for tracing `Segmentation Faults` that cannot be solved in `gdb`
- `htop` - useful resource monitor for tracing CPU usage
- `screen` - very useful at working with multiple files in parallel and running experiments without needing to worry about connection drop

Zookeeper

The first external software that we decided to use in the project is `Zookeeper` [13]. It is a service that provides developers of distributed systems an easy way to store configurations. For the scope of the project one instance was more than enough to keep centralized the configuration of the components of the `RocketSpeed` and also of `Apache Kafka` [14]. To be able to integrate Zookeeper with the rest of the system we had to install a `C` library and use the provided API to retrieve the configurations required, e.g. IP and port of each component, etc..

Chapter 3

RocketSpeed Extensions

3.1 Scalability

3.1.1 Problem

The first major problem encountered while working on the RocketSpeed project was the inability of the system to scale to more than one machine. This comes as a big surprise at first as the major claim of the system is that it scales up to a trillion topics. However, the reason why the open source implementation is not scalable is the fact that the Log Device, the component responsible with message persistence, used by the Facebook developers is not made open-source, as it was internally developed at Facebook.

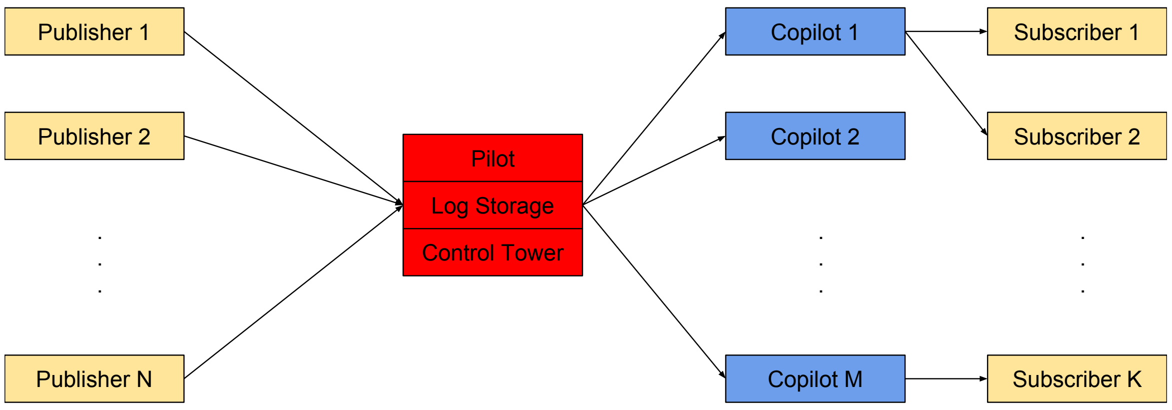


Fig. 3.1 RocketSpeed's open-source architecture

The mock Log Storage provided in the open-source implementation stores all the messages in memory and it does not provide any way of letting other components communicate with it, unless they run in the same program. In order to work properly, the system would need all Pilots and Control Towers to be able to talk to the Log Storage. Hence, decoupling the Log Storage from the Pilot and Control Tower is virtually impossible. So this limits the system to having only one machine running a Pilot, a Log Storage and a Control Tower as in Fig 3.1.

In order to solve the problem of scalability a new implementation of the Log Device needs to be done. To overcome this challenge, three different approaches were considered.

3.1.2 Apache Kafka

Apache Kafka is a distributed, partitioned and replicated log storage system. At a high level the system receives messages from publishers, stores them for a given period of time and sends them to interested consumers.

3.1.3 Topics and Logs

Let us first present the abstractions used by Kafka to store messages. The highest level abstraction is the Topic. Kafka uses Topics as categories with a name to which Producers can publish messages. However, it does not provide any guarantee about message ordering at this level. It further divides each Topic into several Logs, which can be observed in Figure 3.2. It is very important to note that by doing so, Apache Kafka can achieve high scalability, as it can use a very large number of Logs. We will further use the terms Log and Partition interchangeably.

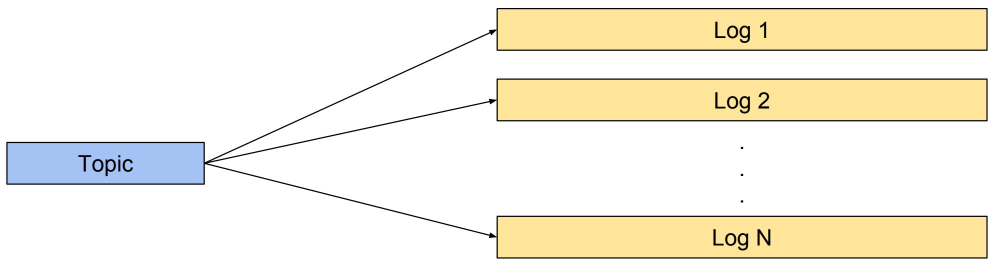


Fig. 3.2 Division of a Kafka Topic into N logs

Each Partition has the property that it is immutable, append-only and totally ordered sequence of messages ordered by time. The partitions are replicated on a configurable number of Kafka servers for fault tolerance. For each partition, there is one server which acts as the leader, while the rest are followers that mirror the behaviour of the leader. If the leader goes down, one of the followers will automatically take its place. Another, valuable property that Topics have is that the user can set up a retention period after which they are purged from the system. We can observe that the system can scale to accommodate any volume of data and the deletition of old data prevents the system from just requiring new partitions to store new data. Moreover, the data size that Kafka stores is not affecting its performance. This fits well with the aim of RocketSpeed of providing a retention period for each namespace.

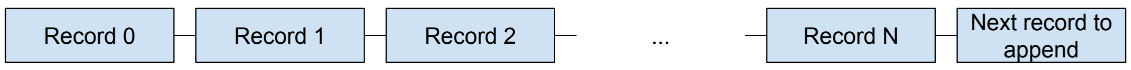


Fig. 3.3 Log Structure

When publishing a message, producers can choose to what topic and partition they will publish the message. They can do this in a round-robin fashion or use any partitioning scheme that they want. Kafka can achieve two types of subscriptions: queuing and publish-subscribe. Queuing means that there is a pool of subscribers and each message will go to only one of them, usually in a round-robin fashion to get an even distribution of the load. In the publish-subscribe paradigm each subscriber will get all messages. However, Kafka offers only one abstraction that can be used to simulate both subscription types.

Kafka provides the concept of consumer group. Each consumer has to belong to a consumer group, but a consumer group is not required to have a minimum number of members. Furthermore, consumer processes are not restricted to live on the same machine. When a message is published to a topic that a consumer group is subscribed to, only one of the consumer instances in the group will receive the message. This leads to a queuing behavior inside each group. So, by making all consumers belong to the same group we obtain a queuing behavior. On the other hand, by making each consumer belong to a different group we get a broadcasting behavior.

Moreover, another interesting feature Kafka provides is that a consumer group can gain a speedup when reading messages from a particular Topic. First of all, let us notice that a Log can be read by at most one subscriber in a consumer group if we want to maintain the guarantee that each consumer group receives messages inside a Log in order. The reason we get this limitation is the fact that messages are send asynchronously, so they can get out of order to different subscribers in the same consumer group. So, the solution is to assign each Log to a reader in the consumer group and thus, each Log will be read in order by the consumer group.

All the above arguments make Kafka a very strong candidate for implementing the Log Storage for RocketSpeed. However, in order to be able to use Kafka to implement Log Storage's interface, a better understanding of how Kafka works, its' API and configuration parameters are required. Thus, it would require a long time to be invested in research and development.

3.1.4 A New Implementation of Log Device

The next solution considered was to develop a new implementation of Log Device from scratch without using any external libraries. As it would mostly be used for testing and not for a real deployment, the idea was to actually keep the messages in memory, but to distribute the topics across several machines by using a hashing function. This would allow the number of topics to increase with the number of machines. Yet, another problem is encountered.

Most of the topics will not have sufficient messages to fill a machine's memory, but there can be a few hot topics that have a large number of messages. These can be co-located by the hashing function on the same machine and together fill up the memory, while if distributed in a different manner to fit in different machines' memory. Even if we were to devise an algorithm to determine how to move around the topics, there would be another obstacle. There is the case that a hot topic will have a large volume of messages such that it does not fit in a machine's memory. This would require a mechanism of splitting a topic across multiple computers.

However, we can observe that the only advantage of implementing Log Device from scratch would be the reusability of parts of the current implementation of the in memory Log Storage, but requires solving problems that Apache Kafka is already capable of solving. Moreover, the above implementation would be suitable for testing, but not for a real deployment. Using Apache Kafka to implement a Log Storage would yield a more useful extension of RocketSpeed that could actually be used in production.

3.1.5 Memcached

The third solution proposed was Memcached. This is a high-performance, distributed caching system. It was originally developed for speeding-up web applications by reducing the number of queries to the database, leading to smaller latencies.

How does it work?

In order to achieve its goal, Memcached groups several servers into a system and then it shares the memory available on each server to the whole system. This way every single server can access the whole memory pool available in the system. Using this structure, Memcached then uses all the servers in the system to store in memory a huge distributed hash table. The system does not use any data structures nor does it make any assumptions about the data size. Thus, it offers a high flexibility regarding the format of the stored data, but it requires that any stored object to be pre-serialized. However, the problematic issue is that when the hash table gets full, any additional insertions will lead to old entries being removed without any notice, as Memcached acts as a Least Recently Used Cache.

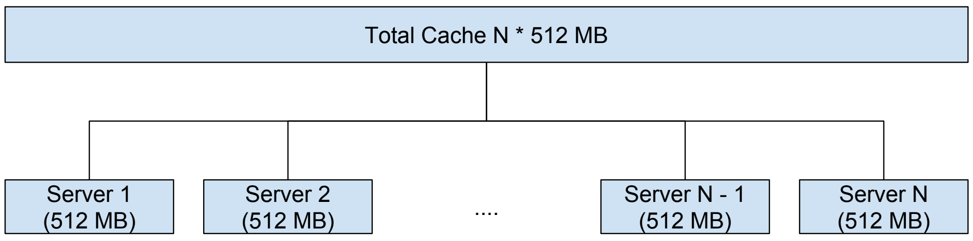


Fig. 3.4 Memcached Overall Structure

For our problem this would give a provisional solution for testing, but without usability in a real-world context, exactly like the previous solution. However, this poses another problem, if just the machines' memory would be used this would require a larger number of machines and moreover, if the total available memory is miscalculated it can lead to message loss without any warning.

3.2 Flow Control

3.2.1 Problem

For a publish/subscribe system it is very important to have a flow control mechanism that prevents the system from being overloaded with messages. However, being in the developing stages, RocketSpeed is not equipped with such a mechanism that would prevent the system from failing. By running a simple experiment in which we send messages continuously without any break, the system goes down just after $10,000$ messages were published.

This comes as a surprise taking into account that in the specifications of the project, RocketSpeed claims that it utilises a Service Level Agreement which lets each tenant register a maximum input rate and output rate.

3.2.2 Smart Flow Control

In order to solve this challenge we decided to device a Smart Flow Control mechanism that would regulate the traffic based on the current work load of the system. In order to moderate the incoming traffic the mechanism would require to be present on the Pilots, while for outgoing traffic it should be present on the Copilots. Fig 3.5 presents a simple case when the Smart Control Flow would decide to reject the message published by Producer N, as the current performance of the Pilot is degrading.

The project will focus on developing and testing the Smart Flow Control mechanism for the Pilot.

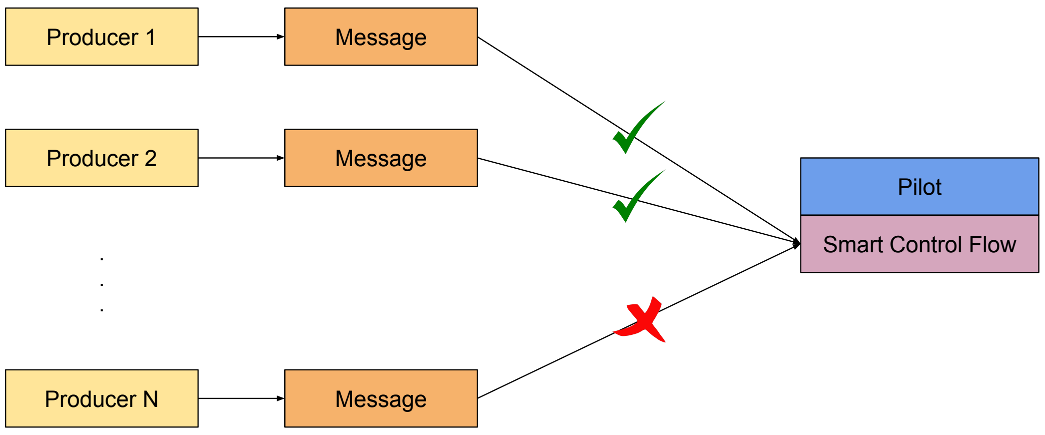


Fig. 3.5 Pilot rejecting message from Producer N

Queuing System

Queuing theory [16] is a branch of applied probability theory which studies systems comprising of a traffic flow and in which queues can appear. Historically, this branch of mathematics has been developed in the context of telephone traffic engineering, but it has application in many real-world problems. For example, how many checkouts should a supermarket have in order to keep customers happy, but also not spend too much money on maintenance or staff? How many toll booths should a bridge have in order to maintain a good traffic flow while avoiding congestions? How many nurses and doctors would be required in the Emergency Department to be able to serve all incoming patients? All these systems can be modelled in the same way and are called queuing systems.

A **queuing system** can be described as a service that clients come to use. They arrive at a given rate to the system, spend some in time the system, then they queue, get serviced and leave the system.

Little's Law

Little's Law [17] states that, under steady steady state conditions, the average number of clients in a queuing system equals the average rate at which clients arrive to the system multiplied by the average time that a client spends in the system.

The formula can be expressed in a mathematical way as follows:

L = the average number of clients in the system

W = the average waiting time for a client in the system

λ = the average number of clients arriving per unit time in the system

Then **Little's Law** becomes: $L = \lambda W$.

This law is remarkably simple and general at the same time. It does not require to know how many servers the system has nor how many queues can the server handle at the same time nor what is the order in which clients are served.

Little's Law is useful for several reasons. First of all, when knowing two of the metrics it offers a very simple way of determining the third one. For example, by controlling the the number of messages in the system, i.e. L, and computing the average latency to serve those messages, i.e. W, we can obtain the throughput of the system, λ. This idea also indicates a safe way of testing RocketSpeed without crashing any of its components. We will further expand on this in the **Evaluation** chapter.

Optimal Throughput

Having these metrics provides us with a good indication of the system's performance based on the work load of the Pilot. Thus in order to develop a Smart Flow Control mechanism that would prevent RocketSpeed components from crashing, a better understanding of these metrics is required.

In Figure 3.6 we can observe the relation between the latency, depicted by the purple graph, and the throughput, depicted by the black graph, with respect to L, the number of in flight messages. As one would expect, increasing the number of in flight messages would result in an increase of the latency, as the system has to process a larger number of messages. However, it is not as trivial to say that the throughput increases, but this is the case as well, as long the Publisher does not push more message than the maximum throughput.

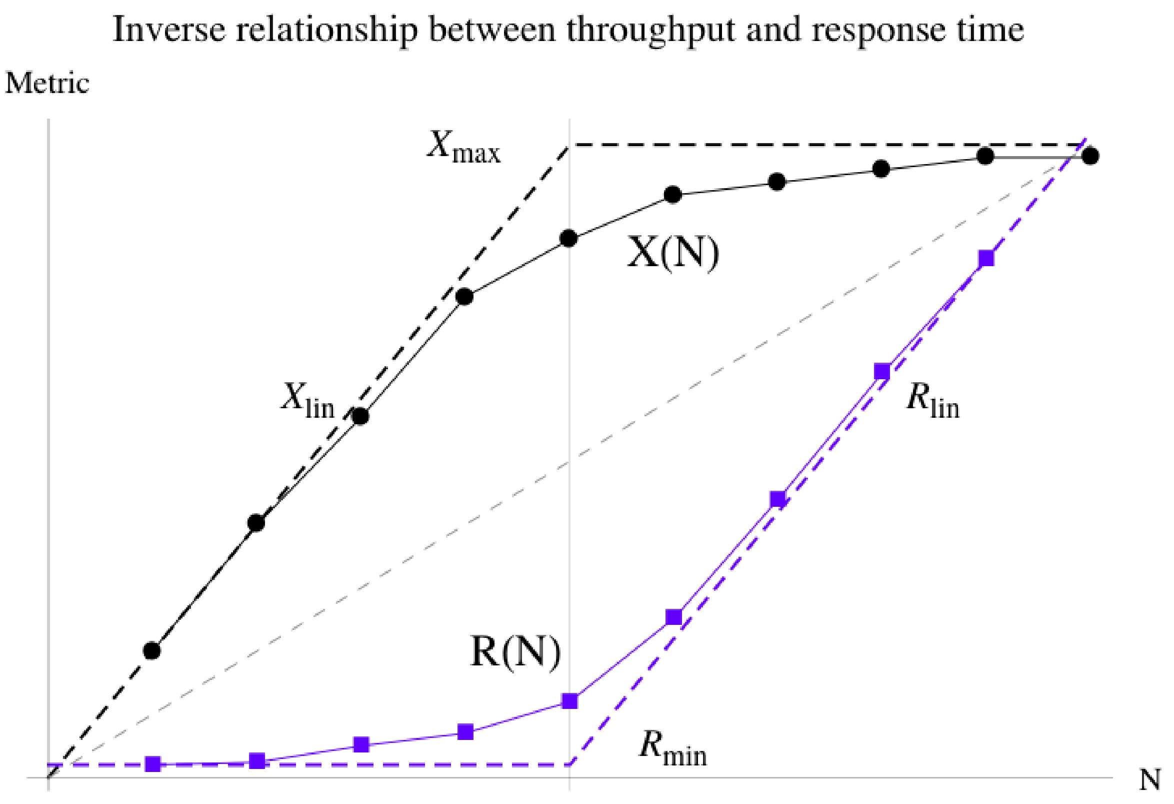


Fig. 3.6 Correlation between latency and throughput [18]

By inspecting the graph we will define an optimal state to be the state when the the in flight number of messages is equal to the point where the vertical line intersects the x axis. That is the point where there is a balance between latency and throughput. We can say that to the left of the vertical line the system's resources are under-utilized, while on the right of the vertical line the system's resources are over-utilized.

Taking into account that in a real world scenario we will not know before hand what is the optimal throughput, we will propose two different algorithms that will try to determine whether the system's current throughput has passed the optimal throughput and if so, reduce the number of in flight messages.

We will discuss the algorithms in the **Implementation** chapter and an evaluation of the performance of the algorithms will be discussed in the **Evaluation** chapter. Let us first make two important observations. By carefully inspecting the graphs in Figure 3.6, we can notice that the latency increases in an exponential fashion up to the optimal point, but then it will have a linear increase with a high slope. On the other hand the throughput will have in the beginning a linear rate of increase with a high slope up to the optimal point, but after that it will keep increasing at a decreasing rate up to the point when the system becomes saturated. These will be the main ideas that the two algorithms will exploit in order to keep the system in an optimal state.

Chapter 4

Implementation

4.1 Connecting Apache Kafka to RocketSpeed

Out of the three proposed solutions Apache Kafka was the one chosen to be implemented for several reasons. First of all, Apache Kafka has the exact behaviour that Facebook's Log Device has, i.e. it is a log storage system. Moreover, Kafka is a scalable solution that already solves problems like partitioning data in small chunks that would ensure that data would fit on machines. Furthermore, plugging Kafka into RocketSpeed would offer a solution to be used in real-world scenarios and not just for testing. The new scalable architecture of RocketSpeed combined with Apache Kafka can be observed in Fig 4.1.

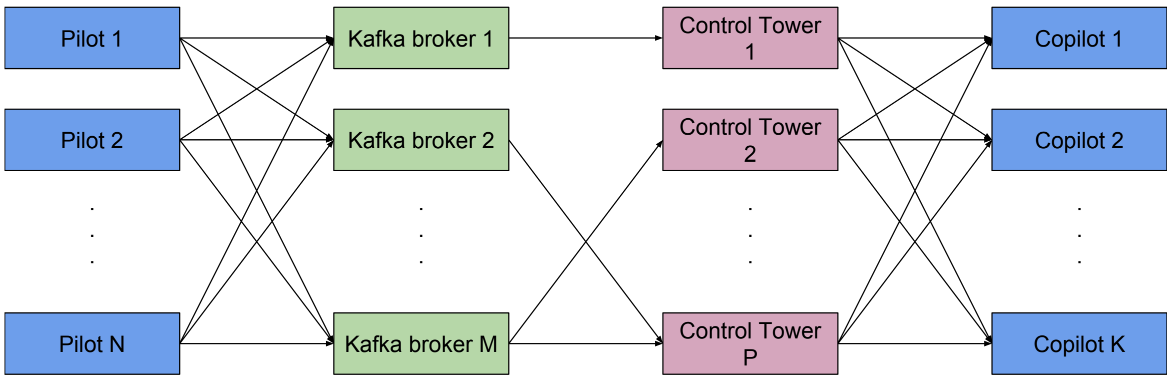


Fig. 4.1 RocketSpeed's new architecture

Before starting to integrate Apache Kafka into RocketSpeed a very important design decision has to be made. RocketSpeed uses the notion of topics for storing messages. However, in order to plug in Apache Kafka as Log Storage the API in Listing 4.1 has to be implemented. This API does not use the notion of topics, but rather relies on the Log Device's log abstraction. This implies that RocketSpeed's logs and topics will need to be matched to Apache Kafka's topics and partitions.

Listing 4.1 Log Storage API

```
class LogStorage {
 public:
  virtual Status AppendAsync(
    LogID id,
    const Slice& data,
    AppendCallback callback) = 0;

  virtual Status FindTimeAsync(
    LogID id,
    std::chrono::milliseconds timestamp,
    std::function<void(Status, SequenceNumber)> callback) = 0;

  virtual Status CreateAsyncReaders(
    unsigned int parallelism,
    std::function<bool(LogRecord&)> record_cb,
    std::function<bool(const GapRecord&)> gap_cb,
    std::vector<AsyncLogReader*>* readers) = 0;

  virtual bool CanSubscribePastEnd() const = 0;
};
```

When solving this design problem, the most important aspect that one must keep in mind is that messages belonging to a RocketSpeed topic must be stored in order. Moreover, more topics are multiplexed into a single RocketSpeed log, as shown in Fig 4.2, which corresponds to a Log Device log. Furthermore, inside a Log Device log all records are ordered and also one such log will reside on a machine.

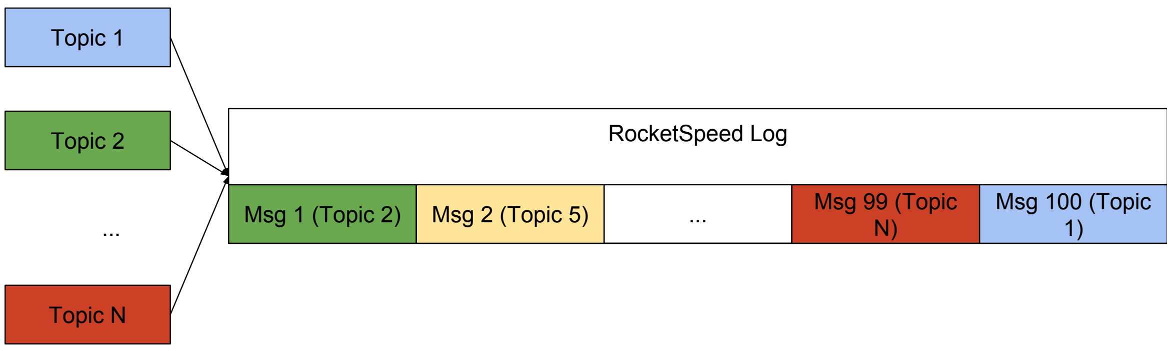


Fig. 4.2 RocketSpeed's log architecture

On the other hand, Kafka gives the following two guarantees: messages inside a Kafka log are ordered and no assumption can be made between messages in different partitions (Fig 4.3 shows that message labelling inside each partition starts from 0) or topics. Having this in mind, we can conclude that the Kafka log is the Kafka component that ensures ordering and has to reside on a machine, i.e. cannot be split.

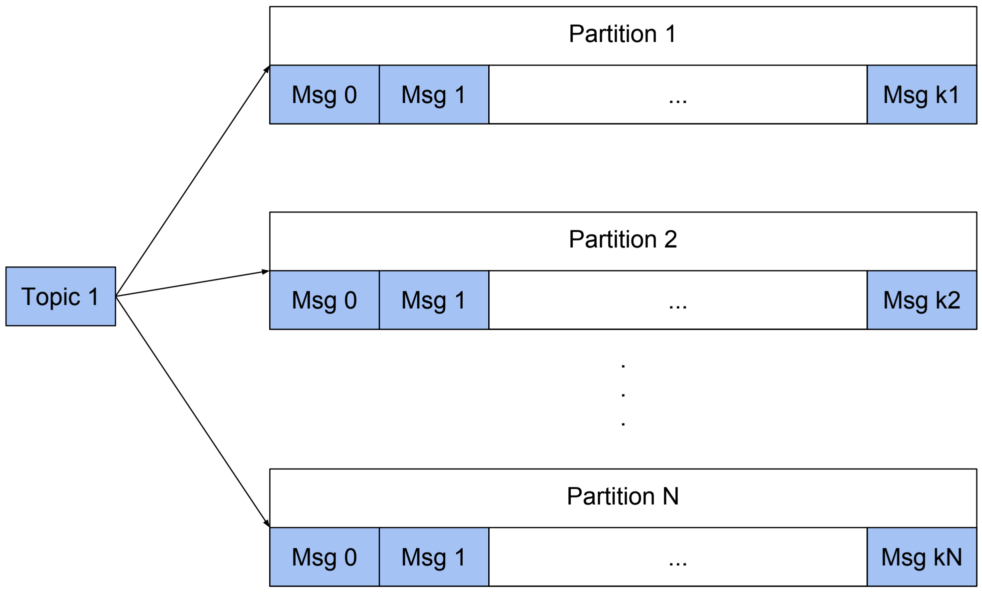


Fig. 4.3 Division of a Kafka topic into partitions

Using the two brief descriptions of RocketSpeed's Log Storage and Kafka, we decided to use the same rationale that RocketSpeed is using with Log Device to store messages.

RocketSpeed logs will be matched to the Kafka partitions, so multiple RocketSpeed topics will be multiplexed into a single Kafka partition. To have a fast matching algorithm from RocketSpeed logs to pairs of Kafka topic and partition, the scheme presented in Listing 4.2 will be used.

Listing 4.2 RocketSpeed log mapping to Kafka topic and log

```
// partitions is the number of partitions that Kafka
// is set up to use
// partitions = the total number of partitions that
// a Kafka topic is using
int32_t LogIDKafka::logIDToKafkaPartition(
    const LogID &id,
    int32_t partitions) {
  RS_ASSERT(partitions > 0);
  return static_cast<int32_t>(id % partitions);
}

std::string LogIDKafka::logIDToKafkaTopic(
    const LogID &id,
    int32_t partitions) {
  return std::to_string(id / partitions);
}
```

In order to manage the transition from RocketSpeed logs and Kafka topic and logs we are going to use the class `LogIDKafka`, whose API is presented in Listing 4.3. This class also provides the bijective functions `KafkaToSequenceNumber` and `SequenceNumberToKafka` to get the bijection between message ids that RocketSpeed uses and the ones provided by Kafka.

Listing 4.3 RocketSpeed translation to Kafka API

```
class LogIDKafka {
 public:
  static int32_t logIDToKafkaPartition(
    const LogID &id,
    int32_t partitions);

  static std::string logIDToKafkaTopic(
    const LogID &logid,
    int32_t partitions);

  static LogID KafkaEntryToLogID(
    const std::string &kafkaTopic,
    int32_t partition,
    int32_t partitions);

  static SequenceNumber KafkaToSequenceNumber(
        int64_t kafka);

  static int64_t SequenceNumberToKafka(
        SequenceNumber seqno);
};
```

4.1.1 Librdkafka

In order to be able to connect to Kafka brokers, publish and read messages a `C/C++` library was needed. For this we chose to use `librdkafka` [19] which is maintained by Magnus Edenhill and is licensed under the 2-clause BSD license.

4.1.2 AppendAsync

This function is responsible for appending incoming messages asynchronously. In order to achieve this, a `MessageLoop` is utilised. This is an important class provided in RocketSpeed's implementation which uses the `libevent` [24] library to handle callback functions. The main focus of this class is to provide a way to have callback function that are invoked when communication between two RocketSpeed components takes place. Moreover, it allows the registration of timed callbacks and it also offers the possibility to register different callbacks that will be enqueued to a list of callbacks to be executed.

`AppendAsync` function uses the latter functionality of `MessageLoop` to register a callback function to publish the received message to a Kafka broker. `MessageLoop` uses multiple threads to execute the callbacks, so in order to take advantage of this parallelism we distribute the callbacks deterministically, using the RocketSpeed topic, to a particular `MessageLoop` queue. The last bit is very important as messages from the same RocketSpeed topic should not end up in different queues, as this can break the message ordering.

Listing 4.4 Callback used to publish message to Kafka broker

```
std::unique_ptr<rocketspeed::Command> cmd(
  rocketspeed::MakeExecuteCommand([this, id, topicName, partition,
  data, cb = std::move(cb)]() mutable {
    std::string errorMessage;

    // Check if there is cached Topic to use for publishing
    if (topics_.find(topicName) == topics_.end()) {
      RdKafka::Topic *topic = RdKafka::Topic::create(
        producer_.get(),
        topicName,
        options_->topicConfig.get(),
        errorMessage);

      if (topic == nullptr) {
        RS_ASSERT(false);
      }
      topics_.emplace(topicName,
                      std::unique_ptr<RdKafka::Topic>(topic));
    }

    char * message = new char[data.size()];
    std::memcpy(message, data.data(), data.size());
    RdKafka::ErrorCode errorCode = producer_->produce(
      topics_[topicName].get(),
      partition,
      RdKafka::Producer::RK_MSG_FREE,
      message,
            data.size(),
            nullptr,
            0,
            cb.release());
        RS_ASSERT(errorCode == RdKafka::ErrorCode::ERR_NO_ERROR);
}));
```

In Listing 4.4 is presented the callback used for publishing a message asynchronously to a Kafka broker by `AppendAsync`. In order to optimize publishing there is only one Kafka `Producer` responsible with publishing messages for all the topics. Furthermore, `KafkaLogStorage` caches the Kafka `Topic` objects used for publishing to a specific topic. By using only one `Producer` the number of threads that gets created is greatly reduced, as each one would use 4 threads, so a large number of unnecessary context switches is avoided. Moreover, the `Topic` objects are lazily created and then are cached for further use. As messages for a specific topic end up in the same `MessageLoop` queue, no data race can appear for `Topic` objects and also `Producer` is also data race free as it uses `Topic` objects in order to publish to different topics.

4.1.3 CreateAsyncReaders

`CreateAsyncReaders`, as the name suggests, is responsible with creation of readers to consume messages for different topics. A new class was introduced, called `KafkaAsyncReader`, which implements the interface of `AsyncLogReader`, the abstract class responsible with reading messages from `Log Storage`.

4.1.4 KafkaAsyncReader

This is the main class responsible with reading the messages from the Kafka brokers inside the Control Tower. Whenever, a subscriber requests to read from a log, an `AsyncReaderConsumeCb` is created and it is assigned an unique id by using a `static std::atomic_int`. As multiple subscribers might request to open the same log from different points, the unique id becomes very important as a reader can open a log only once. So the id will be used to determine the correct reader of the log, i.e. the last one created.

Listing 4.5 Callback used for reading messages

```
std::unique_ptr<rocketspeed::Command> cmd(
    rocketspeed::MakeExecuteCommand([this, id,
    consumeCb = consumers_[id], startPoint,
  endPoint, readerId]() {
    if (!consumeCb.get()) {
      // Someone deleted my consumer, so I will re-open it
      // when I get a chance, to continue reading
      Open(id, startPoint, endPoint);
      return;
    } else if (consumeCb->readerId() > readerId) {
      // Old consumer that was scheduled to run after a new
      // consumer was created
      // Just delete it
      return;
    }

    consumer.reset(RdKafka::Consumer::create());
    topic.reset(RdKafka::Topic::create());

    // start the subscription
    consumer->start();
    while (consumeCb->isRunning() &&
           !consumeCb->isClosed() &&
           timeHasElapsed()) {
      consumer->consume_callback();
    }
    consumer->stop();
    if (!consumeCb->isClosed()) {
      // If did not finish consuming all messages
      // retry from where you stopped
      consumeCb->run();
      startReading(id, consumeCb->lastUnprocessedMessage(),
                   endPoint, readerId);
      return;
    }
  }));
```

In Listing 4.5 is presented a simplified version of the code responsible for reading the messages from the Kafka brokers. The first important part is checking whether someone did not close the reader for the log before this callback got scheduled. The next one is checking if this is not an old consumer that was rescheduled after a new one was created. Without this check, old consumers were never killed and just wasted CPU cycles and time for reading messages that later would not be accepted by the subscriber for being already read. Moreover, the number of consumers for a specific topic would just grow thus making reading slower and slower. However, there are rare cases when a penalty is paid, as we might kill a reader without creating a new one in its place, as that was a reader of a particular subscriber that was behind. However, this is solved by the subscriber, which reissues the subscription for a topic if it does not receive anything for a period of time.

Another decision worth mentioning is that each such callback is allowed to run only for a small period of time, after which it reschedules itself to continue reading. The reason for applying a time limit is the fact that if it starts reading without any stop, then it blocks the `MessageLoop`'s queue of callbacks, making it impossible for the rest of the readers that are in the same queue to read any messages. Thus, every topic will get a fair share of the available resources.

4.1.5 Challenges

Working with `librdkafka` required to invest more time than anticipated to get a good comprehension of the large API that it provides (the header file has almost 1800 lines of code). Moreover, while working with the RocketSpeed project the Makefile provided in the RocketSpeed github repository [8] was used and edited. Thus by introducing the new library in the project the compilation of the project was broken due to the fact that the compilation rules were very strict and all warnings were transformed to errors. In order to fix it, we also patched `librdkafka` such that the errors were solved [20].

Another problem that faced while developing using the `librdkafka` library was the fact that the Kafka components, such as Producers and Consumers, have a large set of configuration parameters that have to be set, but not well documented. For example, when publishing messages, the Kafka broker would send the message id only for the last message published in a batch, while for the rest, would send only 0s. RocketSpeed relies heavily on correct message ids and after discussing with the library's maintainer, we discovered that this was an optimization [21] which could be turned off by setting a flag.

Moreover, while working on the publisher side of Kafka another issue was encountered. The measured optimal throughput of a publisher would be around $80,000$ messages/second when publishing to a single topic. However, when publishing to more than one topic the throughput decreased as low as $2,000$ messages/second. Such a sharp drop in throughput would obviously indicate a problem. After debugging and eliminating the possibility of bugs in the project's implementation, we discussed with Magnus Edenhill, the maintainer of Kafka. After a long analysis of the Kafka logs we discovered that when publishing to a large number of topics the queues for individual topics would not get filled as fast as in the case with one topic. So, the solution was to decrease the timeouts used to flush messages to the Kafka brokers [23].

4.1.6 Apache Kafka Brokers

Starting Apache Kafka brokers is not difficult. First step is to download the software from the official website [15]. However, to be able to start the server a series of parameters need to be set up. Investigating how these parameters need to be set up was the part that took the most. The most important parameters that were common for all brokers are as follows:

- `broker.id`: this has to be unique for each broker and the automated testing script is responsible for assigning a different id for each broker
- `auto.create.topics.enable = true`: this will mimic the behavior of Rocket-Speed's Log Device of creating topics on the fly
- `zookeeper.connect = 146.169.47.3:2181/kafka`: all brokers need to use the same zookeeper instance
- `num.partitions = 10`: in a real-world deployment this number would far bigger. The reason for setting it only to 10 is to route multiple topics to the same log. As described in Apache Kafka's wiki [22] it is more desirable to have a design with a small number of topics and large number of partitions.

4.2 Smart Control Flow

As stated in the **Flow Control** chapter, we will propose two algorithms for implementing a Smart Control mechanism that would automatically regulate the traffic based on latency, throughput and the number of in flight messages.

Both algorithms will make use of a semaphore that will control the maximum number of in flight messages. This valued used by the semaphore will be regarded as $M_{inflight}$. Moreover, a new variable called $delta$ will be introduced, which is responsible for adjusting the value of $M_{inflight}$ by both algorithms. Then the goal of both algorithms is to determine the optimal number of in flight messages and maintain $M_{inflight}$ around that value such that the system would run in an optimal state.

In order to have the latency, throughput and the number of in flight messages available for the algorithms we will first explain how the mechanism determines them.

After every N messages served, the Flow Control mechanism will compute the average serve time, i.e. latency. As the maximum number of in flight messages cannot be used to estimate L, it also tracks the time to receive the N messages. This combined with the value of N give a good estimation of the rate of incoming messages, i.e. λ. Having the latency, i.e. W, and the rate of incoming messages, i.e. λ, it can estimate L by applying Little's Law. Finally, if L is lower than 80% of $M_{inflight}$, then a change in $M_{inflight}$ will not be considered by neither algorithm, as the system currently estimates that it is under-utilizing resources.

4.2.1 The Last Three Points Algorithm

The Last Three Points algorithm is a simple algorithm that tries to determine the optimal throughput by inspecting the last three measured values of the latency and $M_{inflight}$, without taking into account the throughput. The algorithm will start with an initial small value for $M_{inflight}$ and gradually increase it for the first three measurements. Doing this gives us the guarantee that we will always have available data for the last three measurements. We will denote by a, b, c the last three measured latencies in chronological order, i.e. a was measured at t_0, b at $t_0 + 1$ and c at $t_0 + 2$. For the case when the incoming rate of messages is above 80% of $M_{inflight}$ the algorithm will use the following rules to adjust the value of $M_{inflight}$.

1. a, b, c are collinear
   (a) $a < b < c$: the optimal number of in flight messages was passed, so $M_{inflight}$ will be decreased by $delta$
   (b) $a < b >= c$: the latency seems to become constant, so $M_{inflight}$ will be increased by $delta$ to check how big the next increase in latency will be
   (c) $a >= b >= c$: latency seems to be on a decreasing trend, so $M_{inflight}$ will be increased by $delta$ to check how big the next increase in latency will be
   (d) $a >= b < c$: the latency seems to become constant, so $M_{inflight}$ can be increased by $delta$ to check how big the next increase in latency will be
2. a, b, c are not collinear
   (a) $a > b >= c$: there was a sudden decrease in latency, which indicates that the system can handle more traffic, so $M_{inflight}$ will be increased by $2 * delta$
   (b) $a > b < c$: $M_{inflight}$ will remain unchanged as the latency had a fluctuation and will wait for more values to have a better estimation of latency's trend

(c) $a <= b < c$: latency is increasing too abruptly, it might be the case that the maximum throughput sustainable by the system was exceeded, so $M_{inflight}$ will be decreased by $2 * delta$

(d) $a < b >= c$: $M_{inflight}$ will stay unchanged, as the latency has had a fluctuation and will wait for more values to have a better estimation of latency's trend

4.2.2 Adaptive Algorithm

Adaptive algorithm is an algorithm that tries to determine the optimal throughput by doing a more complex analysis of the throughput combined with the latency.

In order to make a better use of the initial observations about the slopes of the throughput and the latency, **Simple Linear Regression** will be used to estimate their slopes. **Simple Linear Regression** is a very cheap algorithm that determines the line which has the least sum of the squared vertical distances between the points and the line. Furthermore, this algorithm will use five points in order to determine the current trends of the two quantities. The reason to choose to analyse last five points is a better overview of both graphs. Moreover, a larger set of points was not chosen, as the two measurements are very likely to have fluctuations and such outlier points to should not affect the computation of the slope for a long period of time.

The following notations will be used in order to make a clearer description of the algorithm:

$S_{throughput}$ = throughput's slope
$S_{latency}$ = latency's slope
$S_{constant} = 0$ (the slope of a constant function)
S_{high} = this will be a threshold indicating a large increase in the slope
S_{medium} = this will be a threshold indicating an increase in the slope, but not very large

For the case when the incoming rate of messages is above 80% of $M_{inflight}$ the following rules will be used to adjust the value of $M_{inflight}$.

1. $S_{throughput} > S_{high}$: this indicates that there was a large increase in the throughput, so the optimal point has not been reached

   (a) $S_{latency} < -S_{high}$: there has been a sudden drop in the latency, so $M_{inflight}$ will be increased by $3 * delta$ (this case should be very rare, e.g. if there is a sudden drop in the number of publishers)

   (b) $S_{latency} < S_{constant}$: the latency is decreasing or is very close to constant, so $M_{inflight}$ will be increased by $2 * delta$

   (c) $S_{latency} < S_{medium}$: the latency is increasing, but not very much, so $M_{inflight}$ will increase by $delta$

   (d) $S_{latency} < S_{high}$: the latency increase is quite big, but not bigger than the upper threshold, so the value of $M_{inflight}$ will remain unchanged to collect further data

   (e) $S_{latency} > S_{high}$: the latency increase is too big, so $M_{inflight}$ will be decreased by $delta$, despite the large increase of the throughput

2. $S_{throughput} < -S_{high}$: this indicates a sudden drop in the throughput, so the value of $M_{inflight}$ will be decreased by $delta$

3. $-S_{high} < S_{throughput} < S_{high}$: the throughput is considered to be constant in this interval, or the changes are negligible

   (a) $S_{latency} < -S_{high}$: there was a sudden decrease in the latency, so $M_{inflight}$ will be increased by $delta$

   (b) $S_{latency} > S_{high}$: there was a sudden increase in the latency, so $M_{inflight}$ will be decreased by $delta$

   (c) Otherwise, $M_{inflight}$ will remain unchanged

A comparison and an evaluation of the two algorithms will be presented in the **Evaluation** chapter.

Chapter 5

Evaluation

A very important part of the project is evaluation. This is the reason why another important part of the project was to develop a testing environment to assess the RocketSpeed's performance with the in memory Log Storage implementation and the one using Apache Kafka. Moreover, the two algorithms devised need to be evaluated in a real-world scenario.

5.1 Test Framework

The RocketSpeed system is composed of many components. On an abstract level there are 5 major components: **Pilot**, **Copilot**, **ControlTower**, **LogStorage**, **Client** (consumer and producer). In order to keep the main files of these components as clean and readable as possible we decided to make a class called `TestFramework`. This is used in each main file of every component to set up a test environment and provide an easy way to create any objects necessary for the above mentioned components.

As part of the testing environment it allows the user to register a `shutdown` function that is responsible for safely ending a test in case of a `SIGINT` or `SIGHUP` occurs. We decided to override these signals as it would provide a safe way to indicate the premature end of a test. `SIGINT` is the equivalent of `CTRL-C` and it is used for command line testing, while `SIGHUP` is used as a signal to mark the end of a test that was launched using an automated testing script. we will further explain about the automated testing script in the **Automated Testing** section.

Another worth mentioning point is the presence of the `start` and `stop` functions. The `start` function is a blocking function responsible for starting all components of the system that were instantiated. On the other hand the `stop` function should be called to indicate the test should ended. It is be called in the `shutdown` function that the user registers to do the clean up of the test.

Listing 5.1 The testing framework API

```
class TestFramework {
 public:
  TestFramework(int argc, char **argv, void (*shutdown)(int));

  Status initLogRouter();

  Status initLogger(const std::string &name);

  Status initKafkaStorage();

  Status initRSLogStorage();

  Status initCopilotClient();

  Status initPilotClient();

  Status initControlTowerClient();

  Status initRocketSpeedClient();

  // blocking method that starts all components instantiated;
  // call this method at the end of the test so that
  // it blocks until it gets the signal from the stop method or
  // the test ends
  virtual void start();

  // method to be used in shutdown to end the test
  void stop();
}
```

5.2 Automated Testing

Since the system comprises of so many components and multiple machines can run the same type of component, the next natural step was to write an automated script that would connect to different machines, run the components, wait for a specific amount of time and then end the test.

To achieve this we decided to use `Python` as scripting language as there is a wide range of packages available for use. However, in order to SSH into different machines and run multiple commands we had to use a `Python` module called `Paramiko`. This module provides an implementation for the SSHv2 protocol and offers a simple API that allows the creation of a SSH connection on top of which we can open an interactive shell on the remote machine. Another `Python` module that we have considered for this job was `pylibssh2`, which would provide the same functionality. However, `Paramiko` seems to be highly more popular and thus has the advantage that there are many examples online, which made development much easier.

Taking into account that there was a large set of machines to connect to and execute different commands, we decided to use an Object Oriented Programming approach and build a class called `ConnectionManager` that handled connection creation, running commands and closing the connections safely.

As we can observe in Listing 5.2, the class provides an elegant API that allows handling of multiple connections to machines that run the same type of components. First we just pass to the constructor a list of the machines that we want to connect to. Then by calling `establish_connection_with_hosts` we create an SSH connection to each machine. Then by calling `make_sessions` we use the SSH connections to open an interactive shell with which we can communicate by using `run_commands` to send a list of commands that have to be executed on each machine.

Having this class allows a very easy handling of all connections and increases readability of the main testing script. This way all that needs to be done in the main testing script is to provide for each component type a list of machines where it should run and a list of commands that have to be run. Another important remark to make is that some of the commands require to be made particular for each machine as they require an id (e.g. each publisher/-subscriber has an id, each Kafka instance requires an id, etc.). For this reason we created another class called `Config` that stores all the fixed commands and helper functions that add unique ids to the commands marked.

Listing 5.2 Connection Manager API

```
class ConnectionManager:

  sessions_ = []
  clients_  = []
  hosts_    = []

  def __init__(self, hosts):
    self.hosts_ = hosts

  # PSW - password used to connect to each host
  def establish_connection_with_hosts(self, PSW):

  def make_sessions(self):

  def stop_clients(self):

  def stop_sessions(self):

  def run_commands(self, commands):
```

5.3 Testing Preamble

Having the above generic testing framework allows us to run any particular test. In this section we will make some general observations that apply to both Publisher and Subscriber testing, while the following two sections describe in detail the actual testing methods that will be used to evaluate the system's performance. The testing paradigm will be the same for all tests, however, we will vary the number of publisher/subscribers, the number of system components, log device type, the number of messages published and the message distribution to different topics.

Each Publisher will be paired with a Subscriber to run on the same machine. In order to achieve this pairing we will assign them the same id. The main reason for doing this is that we want to measure for each Subscriber the time it takes from the moment a Publisher produces a message to the moment it receives it. To be able to measure this time difference we will send dummy messages of predefined sizes which contain the timestamp of the message creation. If the Subscriber was not on the same machine as the Producer, then there would be a time difference between the clocks of the machines, which would potentially corrupt the data as the time differences are of order of tens of milliseconds.

5.4 Subscriber Testing

The Subscriber is interested in measuring the time it takes to receive a message since it was published. As explained above we will consider only messages generated by the co-located Producer. However, even this message sampling is not enough to store all that information in memory. The reason for this is that we want to run tests for longer period of times including up to 1 billion of messages. There are several solutions that can be employed.

The first one is to do sampling over the set of messages of interest. Sampling is the process of selecting a subset of messages and use them to characterize the whole set. This can be easily implemented by taking every 1 in N messages and recording data for it. However, this is prone to recording skewed data as well. For this reason, we decided to develop a solution that would take into account all messages.

The next solution that considered was aggregating every N consecutive messages, record the average and then write it to a file. This solution seemed the best as it required only a small buffer in which every position would hold the data for N messages. The reason for using a buffer is that we have to deal with a large number of messages and using only one variable that would record the data for N messages, then reset to 0 and then continue using it for the next N messages is prone to data races and most likely would produce wrong results. Using a buffer of M elements would mean that we have to reuse the same position in the buffer every $N * M$ messages. Setting N to $50,000$ and M to $10,000$ already gives us a window of $500,000,000$ messages to process until we have to reuse the same position in the buffer for a new measurement. However, this solution has a flow. It assumes that messages come ordered by their message id. This is true when the Subscriber is interested only in a topic. However, when consuming multiple topics, the overall message ordering is not guaranteed. This solution would have worked for any number of messages, however, it needs a fix to record data correctly.

In order to overcome the problem with the message ordering we made a compromise and imposed the restriction that we will not reuse any position in the buffer as we can make no assumptions about the time when all N messages corresponding to a position will have been received. This means that we will be restricted to receiving only $N * M$ messages, but as we showed above, setting N and M as low as $50,000$ and $10,000$ would already give us a large limit of $500,000,000$ messages to record. M can be increased without any problems even to $1,000,000$ which would require $1,000,000 * 8$ bytes $= 8MB$ memory and would allow statistics storage for 50 billion messages.

5.5 Publisher Testing

When testing the Publisher we are interested in the time that it takes a message to be published successfully, i.e. the time difference between the moment the message is created and sent to the moment the Publisher receives the acknowledgement from the Pilot. To measure this it is enough to use, as the Subscriber does, the timestamp the message contains.

The next issue to be tackled was to design a testing strategy. The Producer is supposed to publish messages, but it cannot just fire messages at the system, as it would just overload the system and eventually crash it without the Smart Flow Control mechanism. Even with the Smart Flow Control mechanism installed such a hostile testing strategy was not considered suitable.

However, the algorithms devised for the Smart Flow Control can be easily transformed into a very good testing strategy. At the Pilot level, the Smart Flow Control could measure the incoming rate of messages, i.e. λ, and the average latency it takes to serve a message, i.e. W. In contrast, at the Producer level, the same idea with the semaphore can be used to control L, the in flight number of messages and we can also measure, as explained above, the average acknowledgement time, i.e. W. Moreover, the two proposed algorithms that the Smart Flow Control can be used to control and determine the optimal number of in flight messages that the system can serve.

Listing 5.3 presents a very basic implementation of Little's law combined with the semaphore. By inspecting the code we can observe that the semaphore is doing all the hard work to keep at most `inflight` number of messages in the system. Considering that creating a new message is done in constant time, we can safely assume that at any given point in time there will be the required number of messages in the system, i.e. `inflight`.

Listing 5.3 A simplified implementation of Little's Law

```
uint64_t count = 0;
uint64_t id = 0;
int32_t inflight = 1000;
shared_ptr<Semaphore> sem(new Semaphore(inflight));

while (count < numberOfMessages_) {
  sem->Wait();
  automsg = generateMessageForRandomTopic();
  auto receiveCallback = [sem]() {
    sem->Post();
  };
  publishMessage(msg, id, callback);
  count++;
  id++;
}
```

5.6 Observations

To achieve a good understanding of the system's performance and the behaviour of the two algorithms proposed we decided run several experiments in which the number of topics and message distribution into these topics were varied. The number of topics used for testing will be 1, 16 and 256 respectively. For the message distribution we are going to use Uniform Distribution (Fig 5.1), Normal Distribution (Fig 5.2) and Poisson Distribution (Fig 5.3).

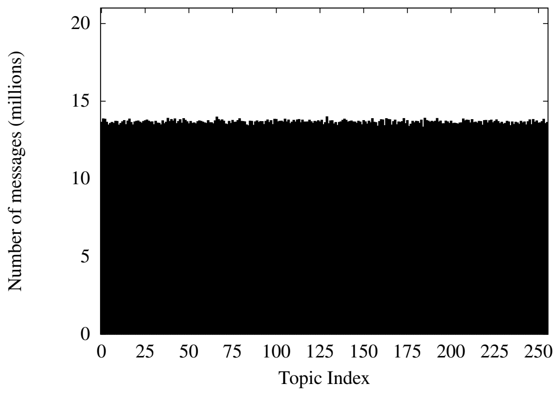


Fig. 5.1 Uniform Distribution of messages

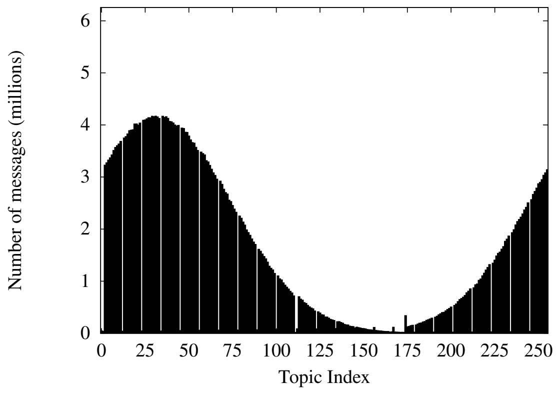


Fig. 5.2 Normal Distribution of messages

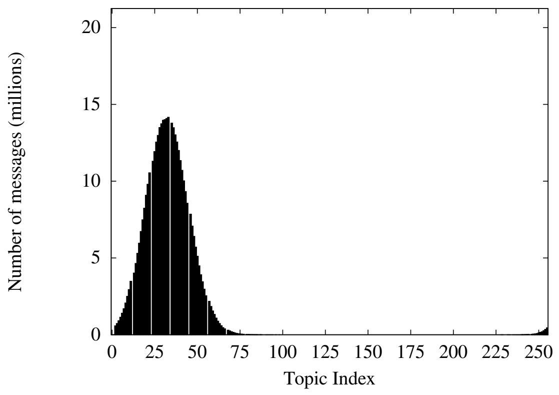


Fig. 5.3 Poisson Distribution of messages

Moreover, for each test we will set up another three parameters: *initial_volume* (the initial number of in flight messages), N (the number of messages to be aggregated) and *delta* (the smallest change in flow to adapt the current volume of in flight messages). These parameters are used by the two algorithms that are controlling the Producers' publishing flow.

To achieve a thorough analysis of RocketSpeed and the two algorithms we will present graphs depicting the number of in flight messages, throughput, latency and relationship between the throughput and the latency with respect to the number of in flight messages. As explained in the **Subscriber Testing**, the statistics for every N messages will be aggregate into a single point. This is why the x axis for the number of in flight messages, throughput and latency will be the number of messages published divided by N.

The graphs presented for the Producer are generated using the data collected from one Producer by running the experiments multiple times and using error bars to indicate the uncertainty of the results. This was really helpful in analysing the data, as the throughput's and latency's plots for each experiment presented many spikes and by using error bars the plots were smoothened. One of the causes is that whenever the in flight number of messages changes, the semaphore used has to be changed with the correct new value. This leads to a situation when for a short period of time two semaphores are counting the in flight messages, i.e. basically allowing a higher number of messages than desired. For this effect to be minimised larger values for N were employed.

In addition, the Kafka publishers have different queues for each topic they publish to. Whenever a queue is filled, it is flushed to the corresponding Kafka broker. However, when a larger number of topics is utilised, the queues do not fill and rely on a timeout to be flushed. The first fix would be to reduce the timeout period to a minimum. However, this leads to a very chaotic behaviour of the system, as it ends up flushing empty queues many times. Whenever this takes place, time is wasted talking to the Kafka broker and the collection of messages may be blocked. In view of finding a balance between small acknowledgement time for publishers and a stable behaviour of Kafka publishers, the timeout was set by trial and error to 100ms.

5.7 In Memory Log Storage Evaluation

In order to test RocketSpeed with in memory log storage the set up presented in Table 5.1 was adopted.

Number of machines	Component
1	Pilot; Control Tower; Log Storage
1	Copilot
2	Publisher/Subscriber

Table 5.1

The reason for running only one of each of RocketSpeed's components is the inability to decouple the Log Storage from the Control Tower, as explained in the **Scalability** chapter. Having this in mind, the next metric that needs to be set is the number of messages each Producer would publish. By running multiple experiments we concluded that the machine that accommodates the Log Storage (together with Pilot and Control Tower) would be able to store $16,000,000$ messages and not experience performance loss. Thus for the following experiments each Producer was assigned to publish $8,000,000$ messages.

5.7.1 The Last Three Points Algorithm

The evaluations in this section was achieved using **The Last Three Points Algorithm** with $initial\_volume$ set to 200, N to $25,000$ and $delta$ to 20.

Producer

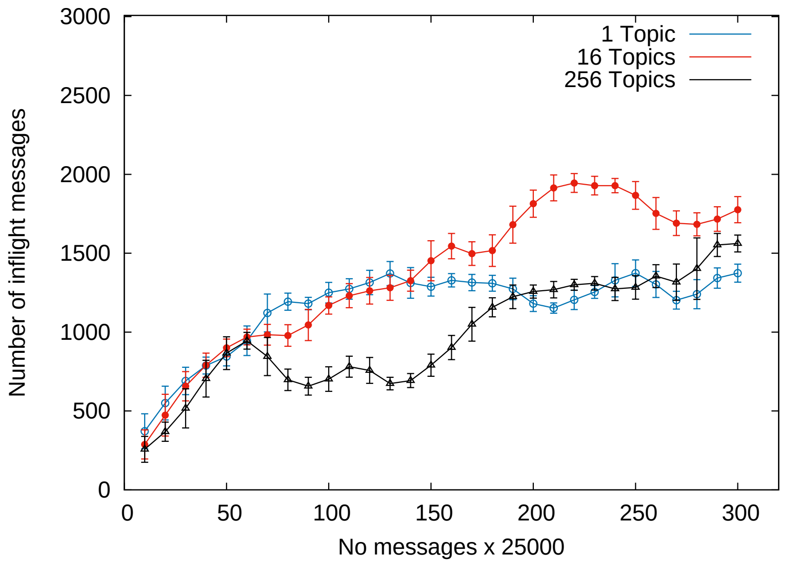


Fig. 5.4 Number of in flight messages; Uniform Distribution

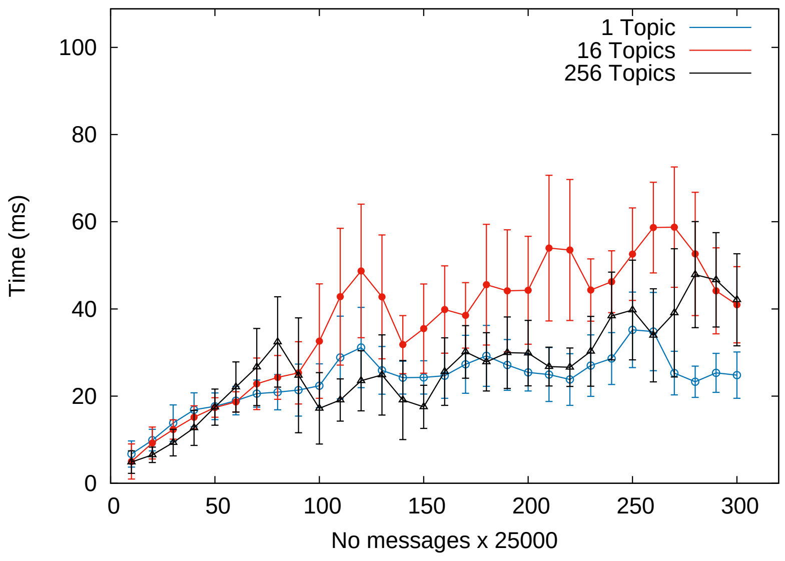


Fig. 5.5 Latency; Uniform Distribution

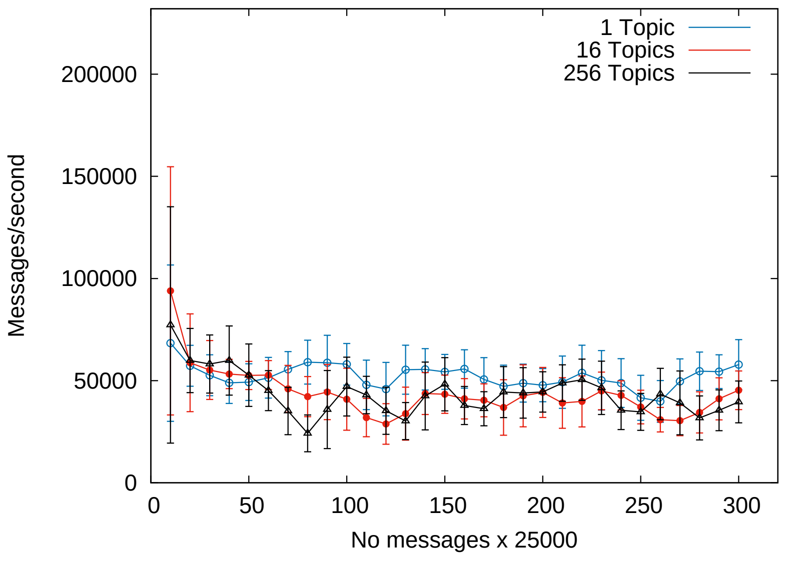


Fig. 5.6 Throughput; Uniform Distribution

Fig 5.4 shows that for a Uniform Distribution of the messages, the number in flight messages is on an ascending trend for all three tests. By inspecting Fig A.1, Fig A.2, Appendix A, we can observe that for 1 and 16 topics this is true as well for the other two distributions. However, it is interesting to notice that in the case with 256 topics this does not hold, the number of in flight messages becoming constant.

Moreover, the latency graphs in Fig 5.5 for 1 and 256 topics are very similar, the one for 16 topics showing a higher variance and bigger values. However, this is not true for the other two distributions, when the graphs for all three cases tend to be close, as can be seen in Fig A.3 and Fig A.4, Appendix A. This shows that the latency tends not to depend too much on the number of topics used.

The throughput graphs presented in Fig 5.6 show that the throughput of the system is not dependent on the number of topics used. The statement holds for the other two distributions as well, as one can observe in Fig A.5 and Fig A.6, Appendix A. For all three distributions the throughput is between $40,000$ and $60,000$ messages/second.

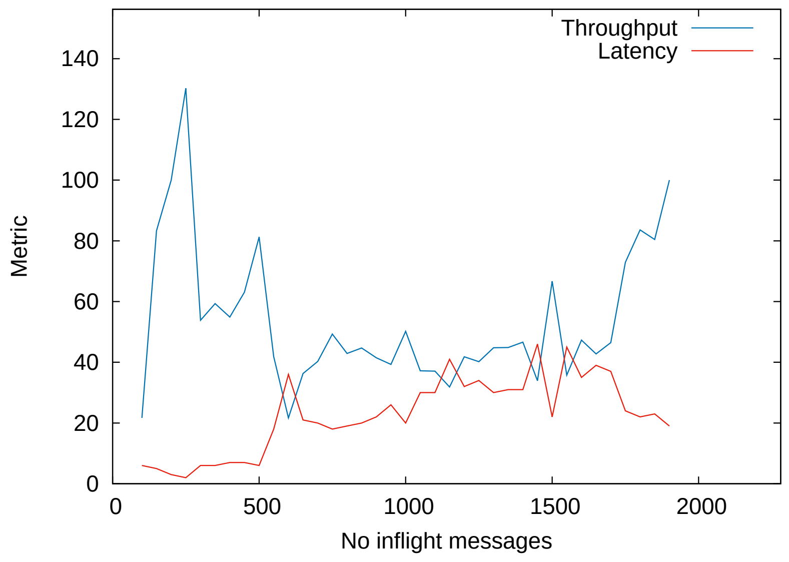


Fig. 5.7 Relation between Throughput and Latency; Uniform Distribution; 256 topics

We can conclude that the algorithm manages to indicate that the throughput of the system is between 80, 000 and 120, 000 messages/second, but it fails to indicate a good candidate for the optimal throughput of the system. The main reason for this incapacity is the low number of messages used for testing. This is backed by Fig 5.7 which does not depict Little's Law as in Fig 3.6, but rather it shows a high variation of the throughput indicating that there was not enough data to eliminate outliers. Coupled with plots in Appendix A.1.1 **Little's Law**, which exhibit a chaotic relation behaviour of the throughput, this is a strong evidence that the number of messages processed was not enough to validate the findings.

Subscriber

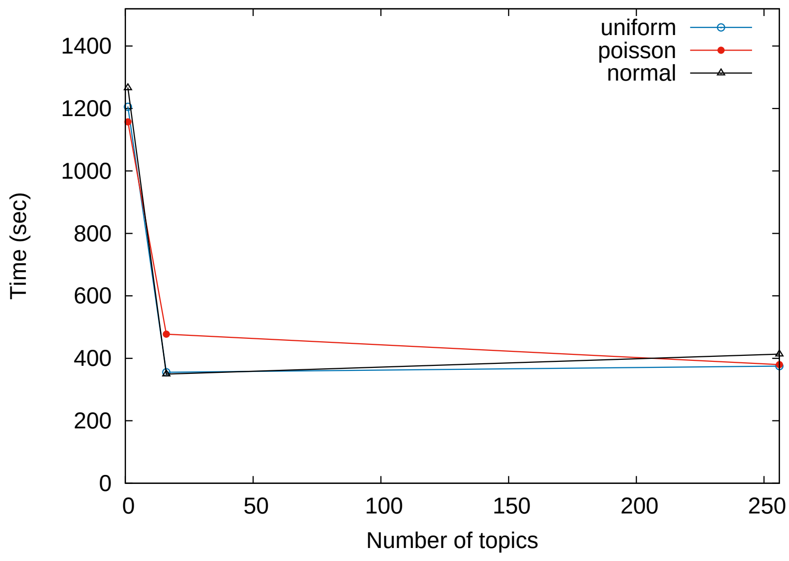


Fig. 5.8 Delivery latency; 99th percentile

Fig 5.8 depicts the 99th percentile of the delay to deliver messages to subscribers. The plot reveals that the latency does not depend on the distribution of messages used. The sharp drop from one topic to 16 topics is a result of the parallelism that RocketSpeed offers while reading from multiple topics. However, the increase to 256 topics incurs a small increase in latency, suggesting that the impact of increasing the number of topics furthermore will not have a major impact on the latency.

5.7.2 Adaptive Algorithm

The evaluations in this section were achieved by using **Adaptive algorithm** with $initial\_volume$ set to 200, N to $25,000$ and $delta$ to 20.

Producer

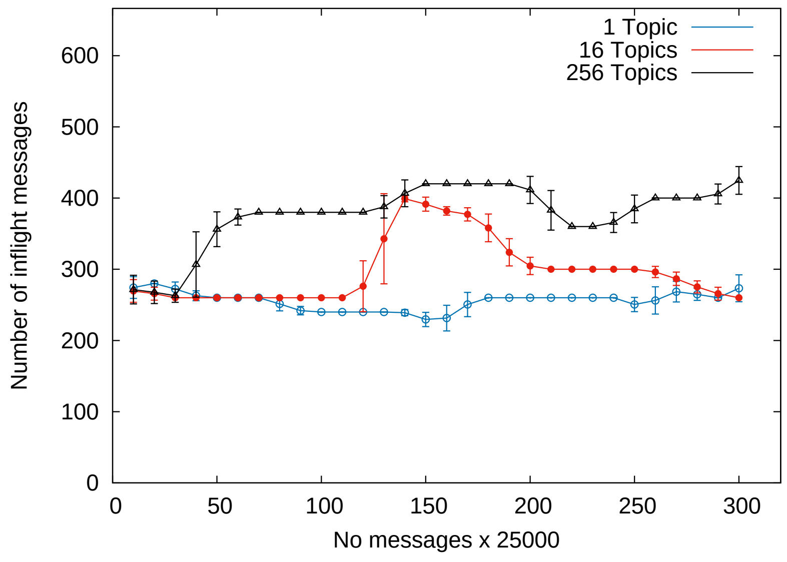


Fig. 5.9 Number of in flight messages; Uniform Distribution

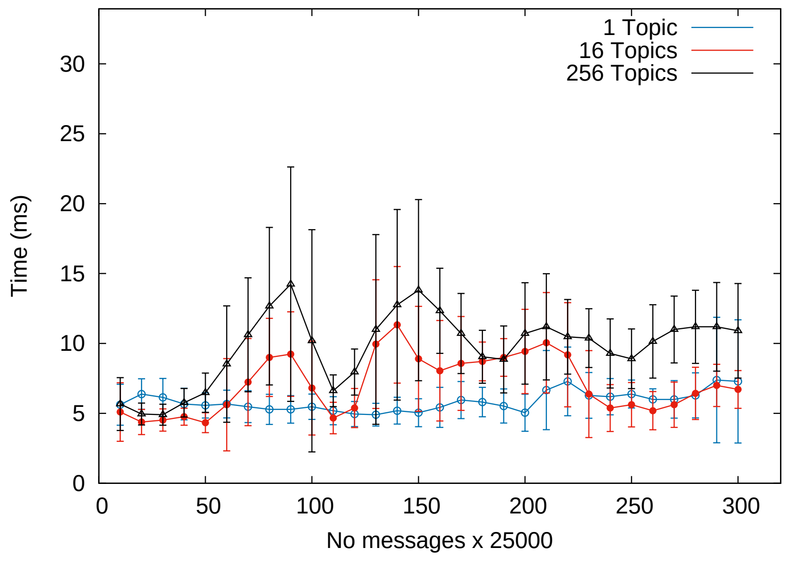


Fig. 5.10 Latency; Uniform Distribution

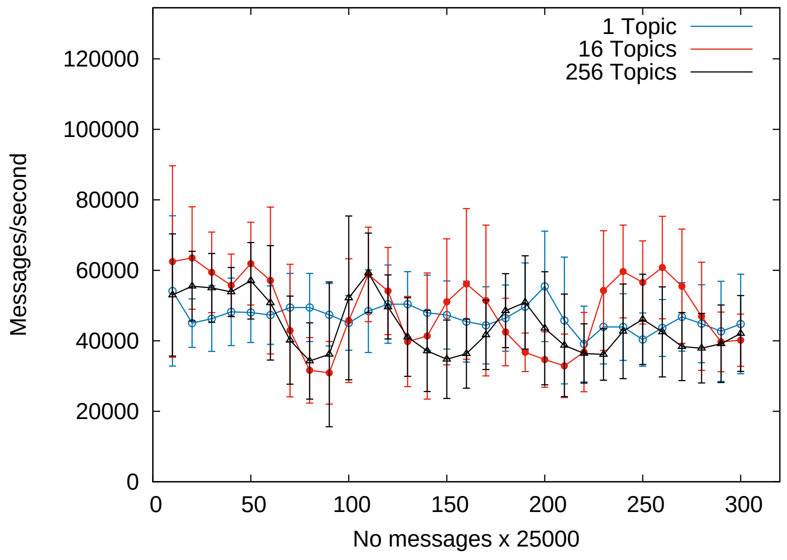


Fig. 5.11 Throughput; Uniform Distribution

Fig 5.9 shows that for a Uniform Distribution of the messages, the number in flight messages is constant for all three cases. Moreover, the number of in flight messages increases with the number of topics. By inspecting Fig A.10 and Fig A.11 in Appendix A, we can observe that the claim holds for the other as well. However, for the Poisson Distribution the number of in flight messages is constant and does not vary too much based on the number of topics.

Moreover, the latency graphs in Fig 5.10 indicate a similar behaviour for the latencies, i.e. they grow with the number of topics. This holds for the other two distributions, as can be seen in Fig A.12 and Fig A.13, Appendix A. However, by analysing the graphs we can detect that the difference in latency is very small.

The throughput graphs presented in Fig 5.11 show that the throughput of the system is not dependent on the number of topics used. The statement holds for the other two distributions as well, as one can observe in Fig A.14 and Fig A.15, Appendix A. For all three distributions the throughput is between $40,000$ and $60,000$ messages/second.

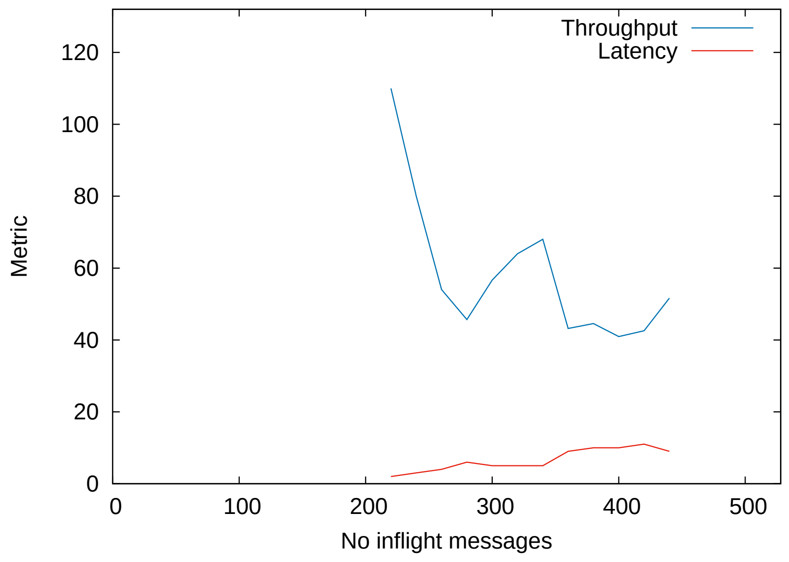


Fig. 5.12 Relation between Throughput and Latency; Uniform Distribution; 256 topics

We can conclude that the algorithm manages to indicate that the throughput of the system is between $80,000$ and $120,000$ messages/second and also indicates a good candidate for the optimal number of in flight messages for the 1 and 16 topics is around 300, while for 256 topics is around 450. However, Fig 5.7 does not depict Little's Law as in Fig 3.6, but rather it shows a high variation of the throughput and reveals the fact that a small set of in flight messages was explored. This indicates that there was not enough data to eliminate outliers. Coupled with plots in Appendix A.1.2 **Little's Law**, which picture a skewed throughput graph and a small set of in flight messages, this is a strong evidence that the number of messages processed was not enough to validate the findings.

Subscriber

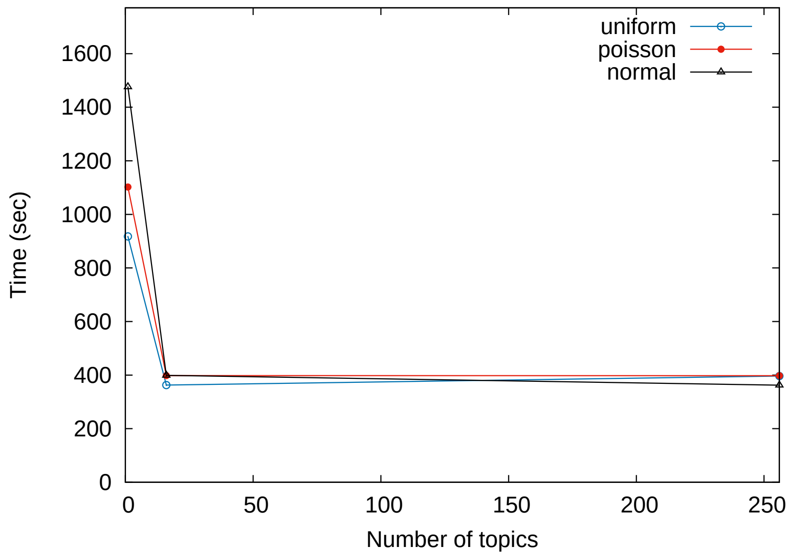


Fig. 5.13 Delivery latency; 99th percentile

Fig 5.13 depicts the 99th percentile of the delay to deliver messages to subscribers. The plot suggests that the latency does not depend on the distribution of messages used. The sharp drop from one topic to 16 topics has the same reason as explained for the previous algorithm and, as in the previous case, the increase of the number of topics incurs a small increase in latency.

The results in both cases are similar and not surprising, as all messages reside in memory on the same machine. So work load is the same, as it is only one machine, and the read time will also be small, as everything takes places in memory.

5.7.3 Algorithms Comparison

From the above mentioned results we can notice that the **Adaptive Algorithm** seems to better at keeping the number of in flight messages constant regardless of the number of topics. Furthermore, the average delay is much lower for all the 3 cases when the number of topics is varied (1 topic: ~5ms < ~35ms; 16 topics: ~7ms < ~30ms; 256 topics: ~15ms < ~20ms). Both of them indicate that the throughput supported by the system seems to be between $80,000$ and $120,000$ messages/second. However, we cannot conclude that any of them manages to find the optimal state, as the graphs depicting the relationship between the throughput and latency are skewed. Additionally, they are a clear indication that not enough messages were processed by the system in order to validate the findings.

In conclusion, even though the **Adaptive Algorithm** seems better when looking at the number of in flight messages and latency, we cannot consider one better than the other, as not enough data was produced to give consistent results. Moreover, we can complete the analysis of the in memory log storage by stating that it is not a good candidate for testing as it is strongly limited by the available memory a machine has.

5.8 Kafka Log Storage Evaluation

For testing RocketSpeed with Kafka as log storage we did not have any limitation on the number of computers that could run a specific component. This is the reason why we chose the following set up, as presented in Table 5.2.

Number of Machines	**Component**
10	Producer/Subscriber
5	Pilot
5	Copilot
4	Control Tower
5	Kafka Brokers

Table 5.2

As this implementation does not suffer from any limitation regarding decoupling of components we have to decide what number of messages will be used for testing. As observed above, $8,000,000$ messages/publisher were not enough to test RocketSpeed with in memory implementation. For this reason we decided that to use $70,000,000$ messages/publisher for the following tests.

5.8.1 The Last Three Points Algorithm

The evaluations in this section was achieved using **The Last Three Points Algorithm** with $initial\_volume$ set to $1,000$, N to $50,000$ and $delta$ to 100.

Producer

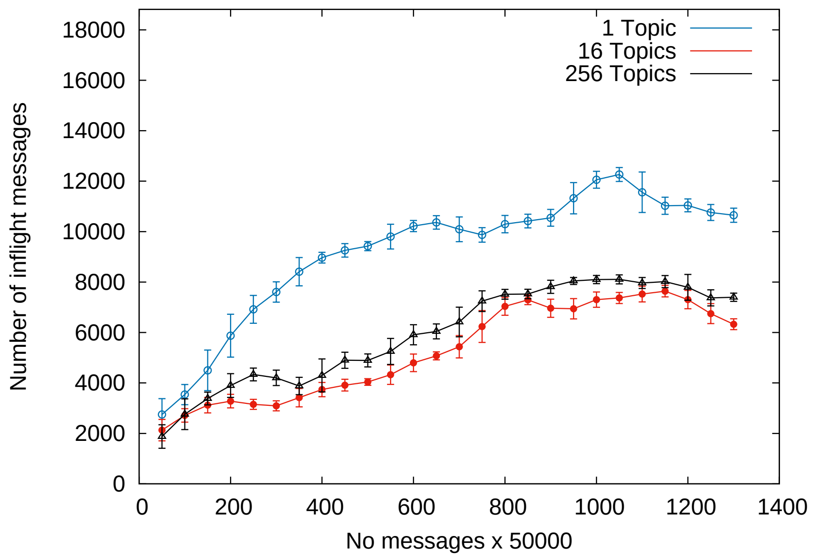


Fig. 5.14 Number of in flight messages; Uniform Distribution

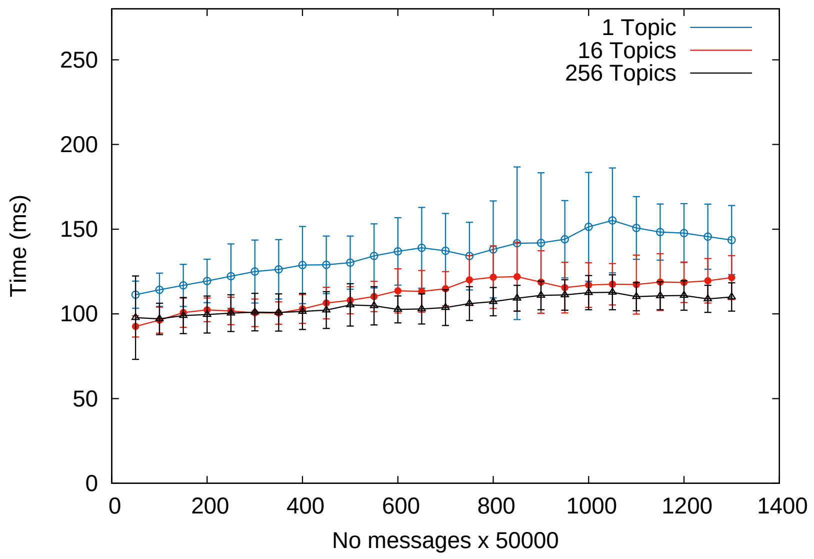


Fig. 5.15 Latency; Uniform Distribution

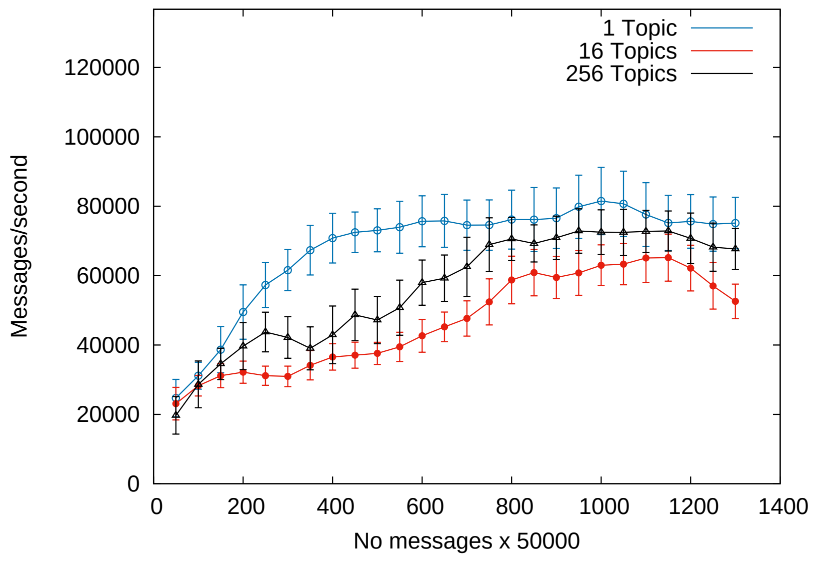


Fig. 5.16 Throughput; Uniform Distribution

Fig 5.14 shows that for a Uniform Distribution of the messages, the algorithm succeeds to increase and then keep constant the number of in flight messages for all three tests. By inspecting Fig A.19 and Fig A.20, Appendix A, we can observe the claim is true as well for the other two distributions. However, it is surprising that for Normal Distribution and Poisson Distribution, when using 256 topics, the number of in flight messages is kept low, around $4,000$.

Moreover, the latency graphs in Fig 5.15 show that the latency has small increases, but is kept low regardless the number of topics. The claim is validated for the other two distributions by the graphs in Fig A.21 and Fig A.22, Appendix A. Altogether, the latency is not dependent on the number of topics or distributions, being situated between 100ms and 150ms.

The throughput graphs presented in Fig 5.16 show that the throughput becomes constant regardless of the number of topics. This statement holds for the other two distributions as well, as one can observe in Fig A.23 and Fig A.24, Appendix A. However, the values for different numbers of topics are not consistent. Only for 256 topics we can observe that throughput is around $40,000$ messages/second. For 1 and 16 topics, the throughput is situated between $60,000$ and $80,000$ messages/second. This suggests that the throughput is lower for a higher number of topics. At the same time, it can also be an indication that there were not enough messages such that the number throughput for 256 topics to reach in the interval of $60,000$ and $80,000$ messages/second.

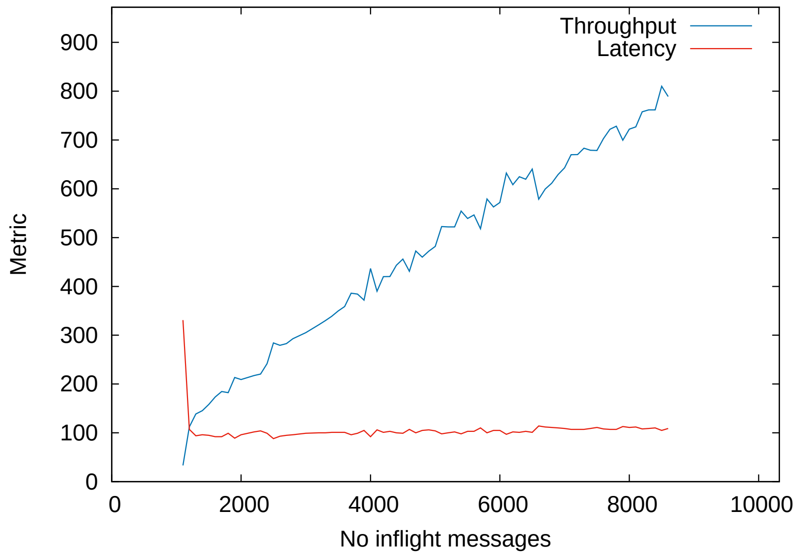


Fig. 5.17 Relation between Throughput and Latency; Uniform Distribution; 256 topics

Fig 5.17 depicts very well the relationship between throughput and latency as in the first half of plot presented in Fig 3.6. This shows that the optimal throughput should be at least $80,000$ messages/second per publisher, thus the optimal number of in flight messages for 256 topics and Uniform Distribution should be at least $9,000$ messages. Together with the graphs in Appendix A.2.1 **Little's Law**, which are very similar to the one in Fig 5.17 we can deduce the same conclusions for all combinations of distributions and number of topics. These plots are a very strong indication that the optimal throughput for a Pilot is around $160,000$ messages/second. Using these plots we can deduce that the algorithm manages to increase the number of in flight message just for 1 and 16 topics towards the optimal number, while for 256 it does not. This is an indication that the algorithm does not work very well for larger number of topics, or simply the number of messages should have been larger for a larger number of topics.

Subscriber

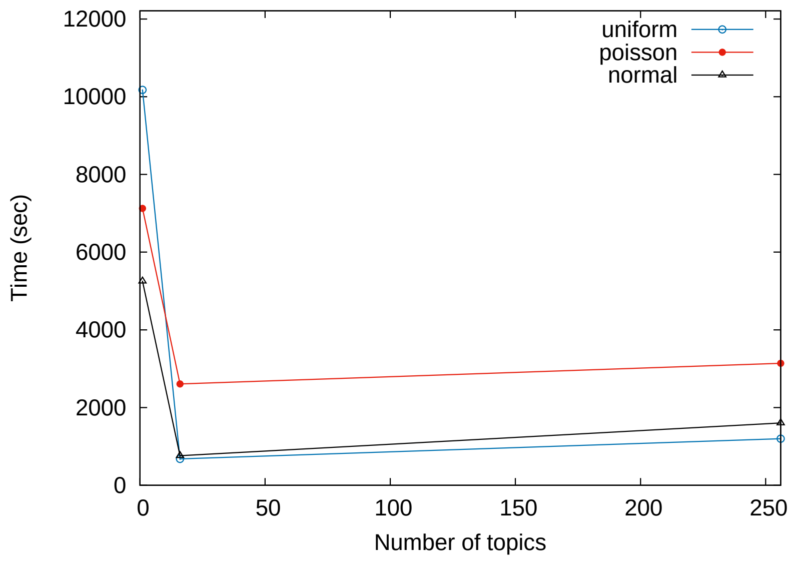


Fig. 5.18 Delivery latency; 99th percentile

Fig 5.18 depicts the 99th percentile of the delay to deliver messages to the subscribers. The plot reveals that the latency increases when a skewed distribution is used. This is expected as, unlike for the in memory log storage, the topics get distributed on different machines. Thus, by using a skewed distribution, we expect not to benefit from the parallelism offered by Kafka brokers. The sharp drop from one topic to 16 topics is a result of the parallelism that RocketSpeed offers, as explained for the in memory log storage. Equally important is the fact that by increasing the number of topics the penalty that has to be paid is small.

5.8.2 Adaptive Algorithm

The evaluations in this section were achieved by using **Adaptive algorithm** with $initial\_volume$ set to $1,000$, N to $50,000$ and $delta$ to 100.

Producer

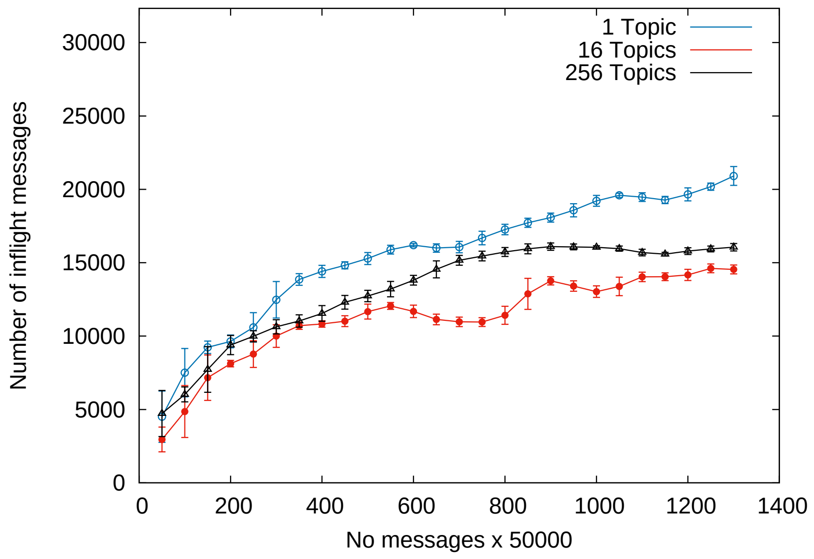


Fig. 5.19 Number of in flight messages; Uniform Distribution

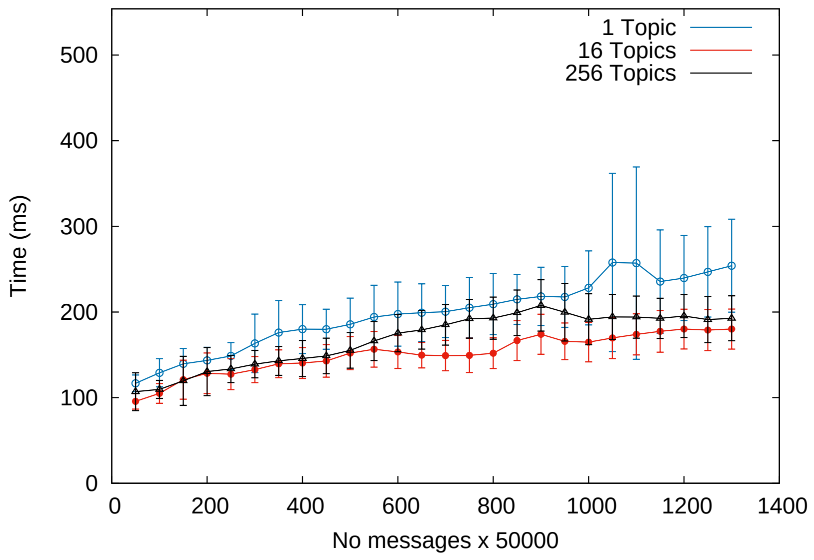


Fig. 5.20 Latency; Uniform Distribution

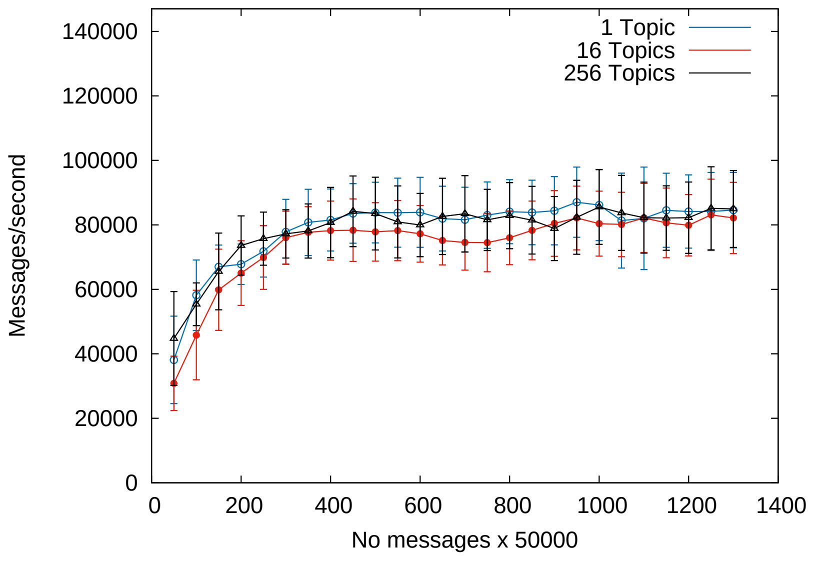


Fig. 5.21 Throughput; Uniform Distribution

Fig 5.19 shows that for a Uniform Distribution of the messages, the number in flight messages becomes constant, in all three cases. This claim is supported by graphs in Fig A.28, Fig A.29, Appendix A. However, for the other two distributions the number of in flight messages tends to be lower and converge towards 15, 000 messages/second.

The latency graphs in Fig 5.20 indicate a similar behaviour for the latencies, i.e. they are constant. This holds for the other two distributions, as can be seen in Fig A.30 and Fig A.31, Appendix A. This behaviour is expected as the in flight number of messages tends to become constant, the latency becomes stable as well. Moreover, the latencies have similar values in all three cases in the interval [100ms, 200ms], suggesting that the number of topics does not influence this metric.

The throughput graphs presented in Fig 5.21 show that the throughput of the system is not dependent on the number of topics used. The statement holds for the other two distributions as well, as can be observed in Fig A.32 and Fig A.33, Appendix A. For all three distributions the throughput is between 70, 000 and 80, 000 messages/second and stays constant.

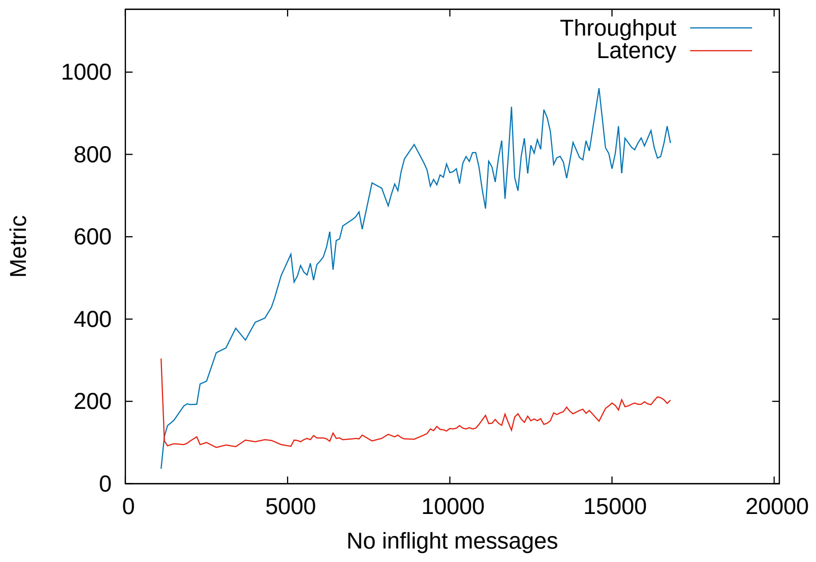


Fig. 5.22 Relation between Throughput and Latency; Uniform Distribution; 256 topics

Fig 5.22 depicts very well the relationship of the throughput and the latency as in Fig 3.6. By analysing the growth of throughput and latency, we can conclude that it offers solid evidence that the optimal throughput for a Publisher is between $70,000$ and $80,000$ messages/second. This hypothesis is strongly supported by the graphs in Appendix A.1.2 **Little's Law**. Moreover, those plots also illustrate that the optimal number of in flight messages for a Publisher is between $10,000$ and $12,000$. Looking back at the evolution of in flight messages, the algorithm does not manage to keep the number of messages in this interval, but rather around $15,000$.

Subscriber

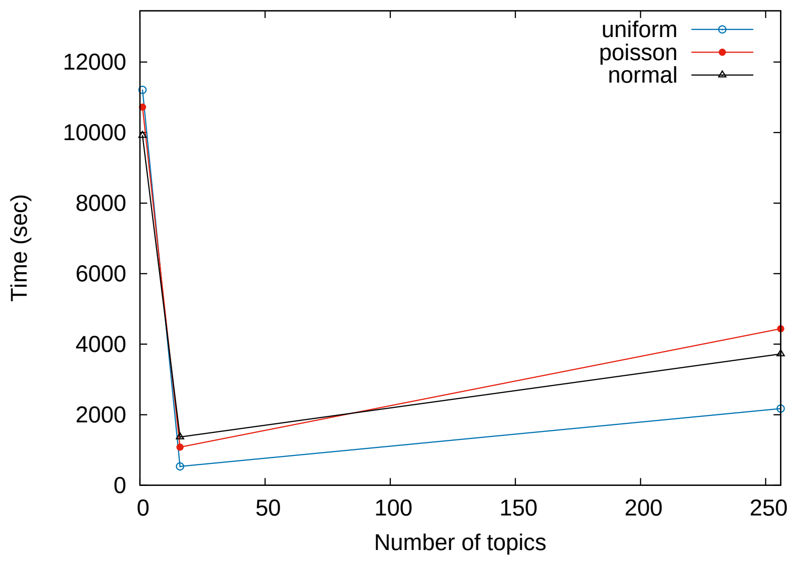


Fig. 5.23 Delivery latency; 99th percentile

Fig 5.23 depicts the 99th percentile of the delay to deliver messages to subscribers. Unlike the previous three Subscriber plots, this shows a high increase in the overall delay. Additionally, as the number of topics increases, the delay increases at a higher rate than for the other algorithm. This is due to the high throughput that **Adaptive Algorithm** promotes.

5.8.3 Algorithms Comparison

The Last Three Points Algorithm indicates that the optimal number of in flight messages should be greater than $9,000$, while the **Adaptive Algorithm** suggests that it should be between $10,000$ and $12,000$. Correlating, these two conclusions together with the plots depicting **Little's Law**, we can safely conclude that the latter algorithm indicates the correct interval for the optimal number of in flight messages. However, neither of them manages to maintain this metric in that interval.

As a result of keeping the number of in flight messages lower than the optimal number, **The Last Three Points Algorithm** preservers the latency for all tests between 100ms and 150ms. On the other hand, the **Adaptive Algorithm** keeps it between 100ms and 200ms.

By inspecting these numbers we can expect that in an optimal state the system would offer an acknowledgement delay of 150ms.

Furthermore, as the above numbers indicate, **The Last Three Points Algorithm** limits the throughput lower than the optimal throughput, especially for 256 topics. In contrast the **Adaptive Algorithm** carries the throughput slightly above the optimal rate.

In conclusion, both algorithms manage to make the number of in flight messages stable, thus protecting the system from overloading. However, neither manages to keep it around the optimal value. The **Adaptive Algorithm** seems to be a better choice as, even though it uses a larger number of in flight messages, it manages to keep the throughput high throughout all tests and the penalty paid for latency is relatively small, around 50ms. As in none of the tests there was a high increase in latency, it shows that the algorithm bounded the number of in flight messages below the point where the system would become over saturated, which makes it a better fit for the Flow Control mechanism as it manages to prevent the system from crashing and also ensures a high throughput.

Chapter 6

Conclusion

Overall we consider the project a success. First of all, we investigated the architecture of the open-source implementation of RocketSpeed and discovered a major drawback, unscalability. Furthermore, we proposed three distinct solutions to fix the issue and chose to plug Apache Kafka into RocketSpeed to provide a scalable architecture for real-world deployments.

Secondly, we analysed Little's Law and exploited it to develop a Smart Flow Control mechanism. For this we devised and implemented two new algorithms that do a thorough analysis of the system's performance by computing the instant throughput, serve latency and number of in flight messages. Moreover, we defined what is the optimal state in which the system should run. By analysing throughput and latency with respect to the number of in flight messages we measured the optimal throughput of a Pilot to be between $140,000$ and $160,000$ messages/second. Furthermore, this result helped us to conclude that the **Adaptive Algorithm** is a better fit than **The Last Three Points Algorithm**, as it allows the system to achieve a higher throughput without paying a big price in latency, only 50ms. Moreover, it manages to reach the optimal throughput for all tests and almost 5x faster than the other algorithm.

6.1 Future Work

In this section we will discuss future improvements that can further enhance RocketSpeed.

- Further investigate the scalability of the system by running multiple experiments on a larger set of machines.
- Perform an analysis on how the message size affects the throughput of the system.

- Devise a distributed caching algorithm at the level of Copilots and thus, trade in duplication of messages in different caches for lower delivery latencies. A suggested topology is presented in Fig 6.1.
- Analyse and improve the topology between Copilots and Control Towers, which currently is a full mesh, in order to achieve greater scability.
- Conceive a topic discovery algorithm by possibly adding metadata to the topics. The current design does not store any information about the topics, thus allowing an infinite number of topics to be added to the system. At the moment a Subscriber needs to know the exact name of the topics that it is interested in, hence a query system would be useful.

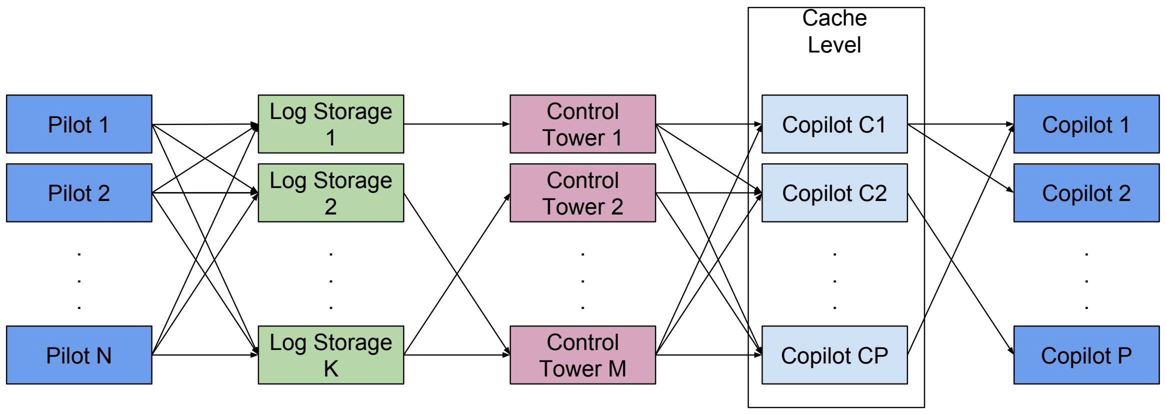


Fig. 6.1 Improved RocketSpeed topology with a caching layer of Copilots

References

[1] http://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/ [online: accessed on 14/06/2016]

[2] http://www.statista.com/statistics/277958/number-of-mobile-active-facebook-users-worldwide/ [online: accessed on 14/06/2015]

[3] https://github.com/facebookarchive/scribe/wiki [online: accessed on 14/06/2016]

[4] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, and Kowshik Prakasam, Facebook; Robbert van Renesse, Cornell University; Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie. ”Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services”. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*, Oakland, CA, USA, NSDI, pages 351-366, May, 2015.

[5] https://www.facebook.com/notes/facebook-engineering/wormhole-pubsub-system-moving-data-through-space-and-time/10151504075843920 [online: accessed on 14/06/2016]

[6] Ming Li, Fan Ye, Minkyong Kim, Han Chen, Hui Lei. ”BlueDove: A Scalable and Elastic Publish/Subscribe Service”. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, Anchorage, AK, IEEE, pages 1254-1265, May, 2011.

[7] Atul Adya, Gregory Cooper, Daniel Myers, Michael Piatek. ”Thialfi: A Client Notification Service for Internet-Scale Applications”. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, pages 129-142, 2011.

[8] https://github.com/facebookexperimental/RocketSpeed/ [online: accessed on 14/06/2016]

[9] https://aws.amazon.com [online: accessed on 14/06/2016]

[10] Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie, Mema Roussopoulos, and Matt Welsh, ”Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds”, *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 2007)*, Cambridge, MA, USA, NSDI, pages 29-42, April, 2007.

[11] Raphaël Barazzutti, Thomas Heinze, André Martin, Emanuel Onica, Pascal Felber, Christof Fetzer, Zbigniew Jerzak, Marcelo Pasin, Etienne Rivière. ”Elastic Scaling of a High-Throughput Content-Based Publish/Subscribe Engine”. In *Proceeding ICDCS 2014 Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, IEEE, pages 567-576, 2014.

[12] https://en.wikipedia.org/wiki/Service-level_agreement [online: accessed on 14/06/2016]

[13] https://zookeeper.apache.org [online: accessed on 14/06/2016]

[14] http://kafka.apache.org [online: accessed on 14/06/2016]

[15] http://kafka.apache.org/documentation.html#quickstart [online: accessed on 14/06/2016]

[16] Robert B. Cooper. ”Introduction to Queuing Theory”, North Holland, 1981.

[17] John D.C. Little and Stephen C. Graves. ”Little’s Law”. URL: http://web.mit.edu/sgraves/www/papers/Little’s%20Law-Published.pdf [online: accessed on 14/06/2016]

[18] http://perfdynamics.blogspot.co.uk/2009/08/bandwidth-and-latency-are-related-like.html [online: accessed on 14/06/2016]

[19] https://github.com/edenhill/librdkafka [online: accessed on 14/06/2016]

[20] https://github.com/edenhill/librdkafka/pull/605 [online: accessed on 14/06/2016]

[21] https://github.com/edenhill/librdkafka/issues/658 [online: accessed on 14/06/2016]

[22] https://cwiki.apache.org/confluence/display/KAFKA/FAQ#FAQ-HowmanytopicscanIhave? [online: accessed on 14/06/2016]

[23] https://github.com/edenhill/librdkafka/issues/684 [online: accessed on 14/06/2016]

[24] http://libevent.org [online: accessed on 14/06/2016]

Appendix A

Plots

A.1 In Memory Log Storage

A.1.1 The Last Three Points Algorithm

In flight messages

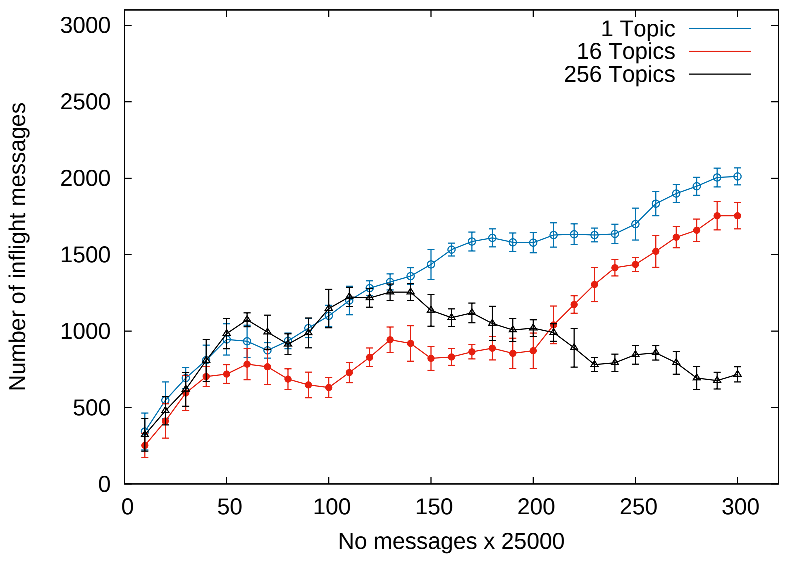


Fig. A.1 Number of in flight messages; Normal Distribution

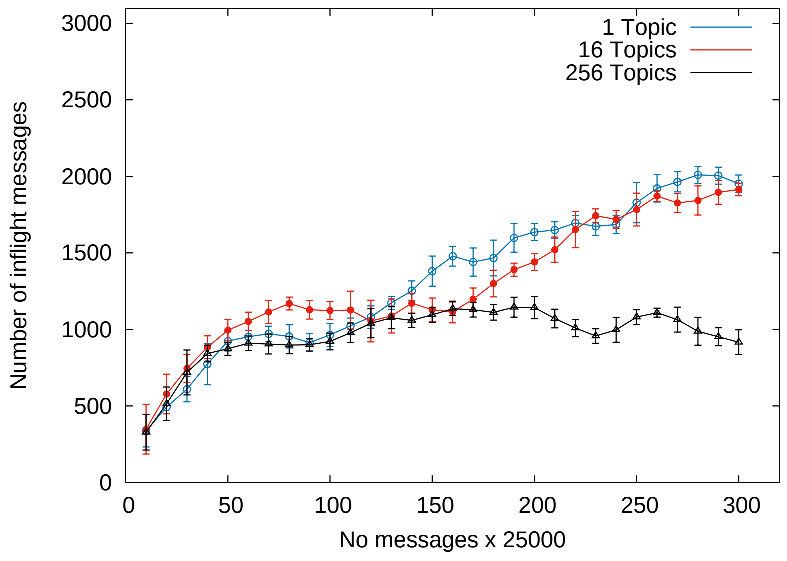


Fig. A.2 Number of in flight messages; Poisson Distribution

Latency

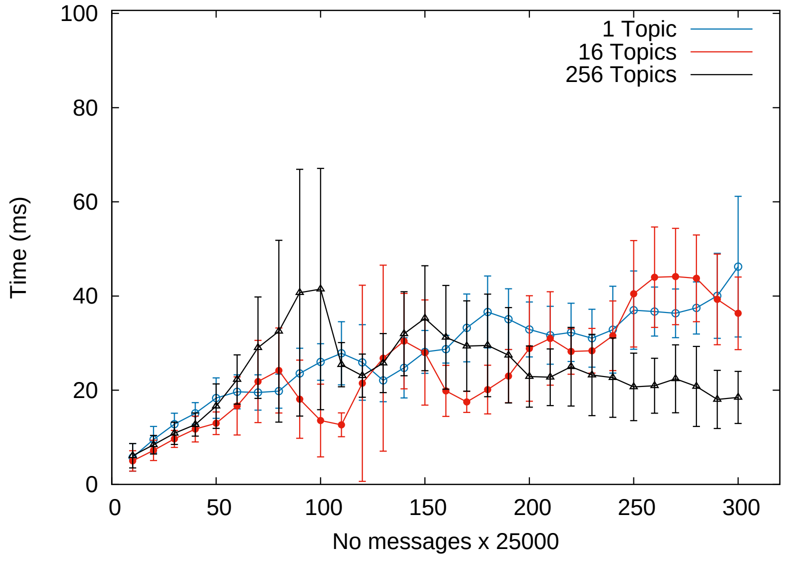


Fig. A.3 Latency; Normal Distribution

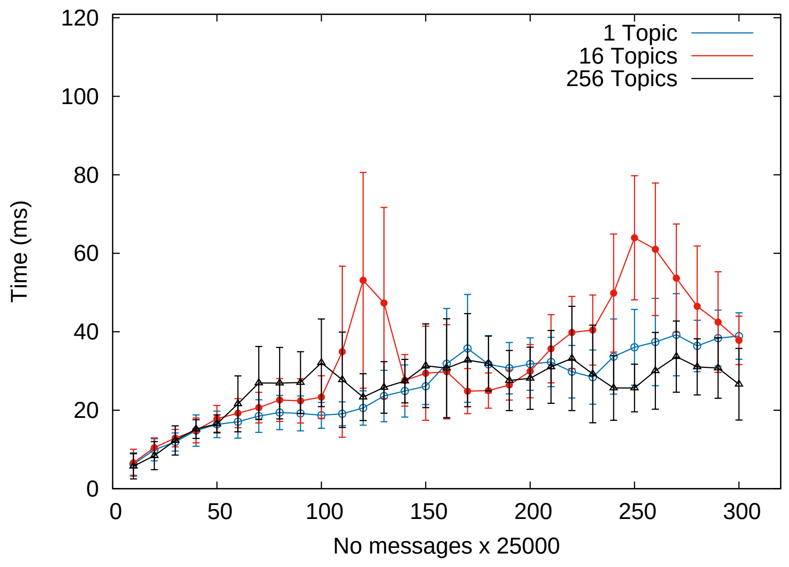


Fig. A.4 Latency; Poisson Distribution

Throughput

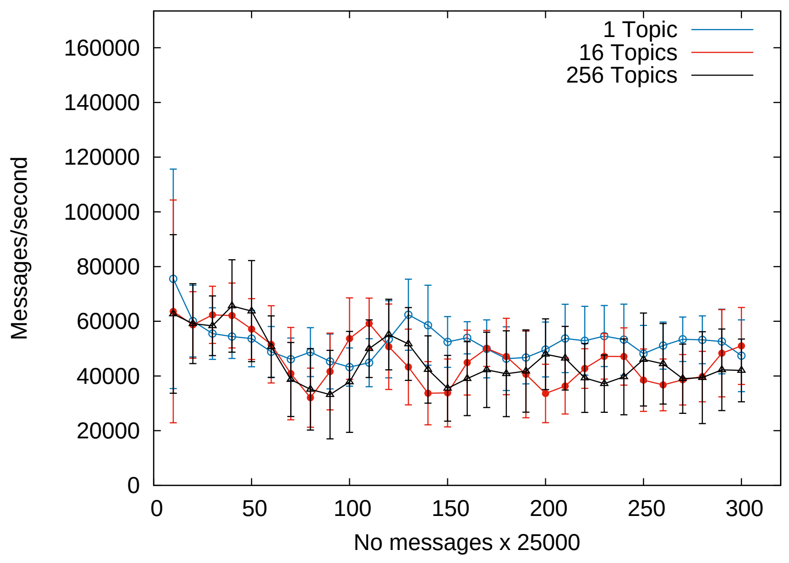


Fig. A.5 Throughput; Normal Distribution

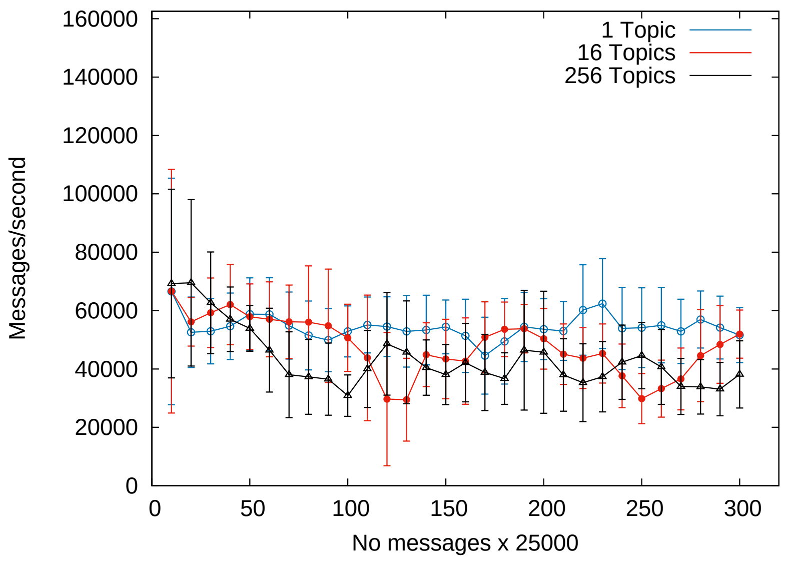


Fig. A.6 Throughput; Poisson Distribution

Little's Law

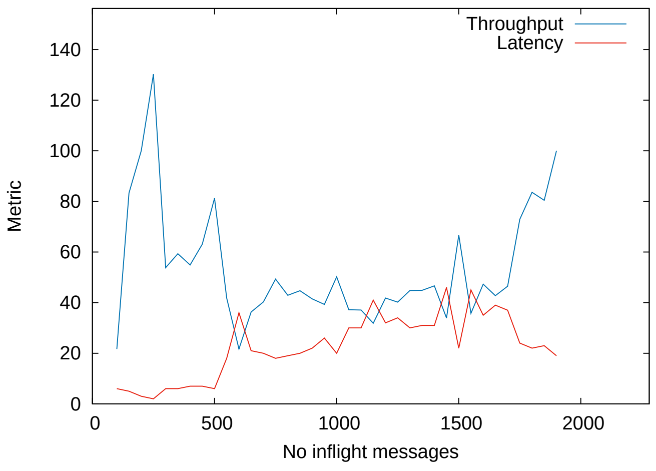


Fig. A.7 Relation between Throughput and Latency; Uniform Distribution; 256 topics

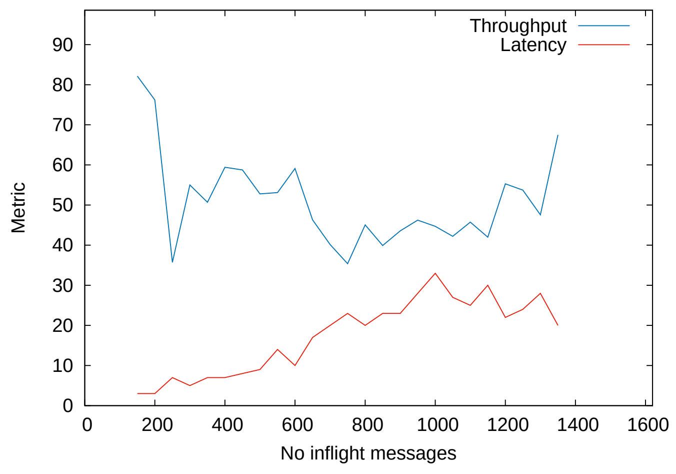


Fig. A.8 Relation between Throughput and Latency; Normal Distribution; 256 topics

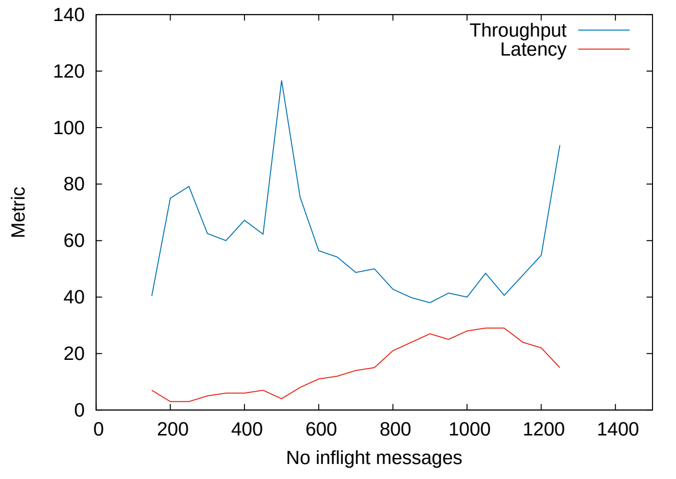


Fig. A.9 Relation between Throughput and Latency; Poisson Distribution; 256 topics

A.1.2 Adaptive Algorithm

In flight messages

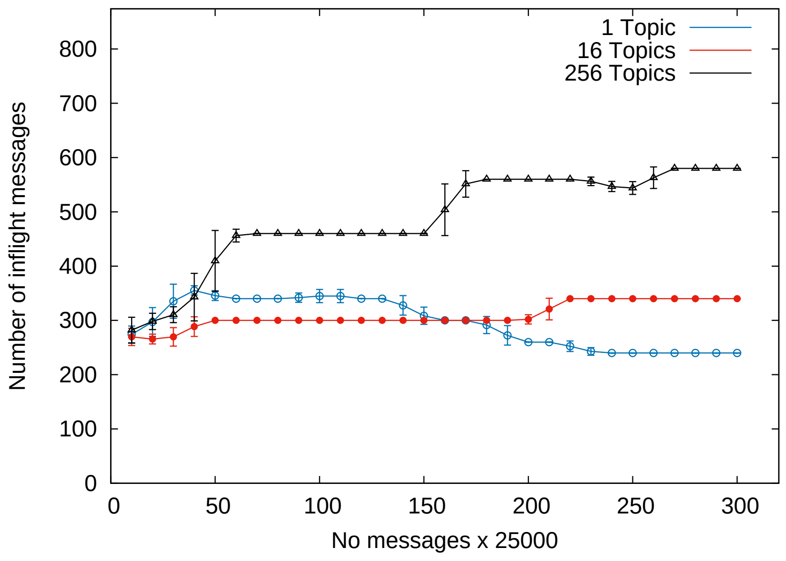


Fig. A.10 Number of in flight messages; Normal Distribution

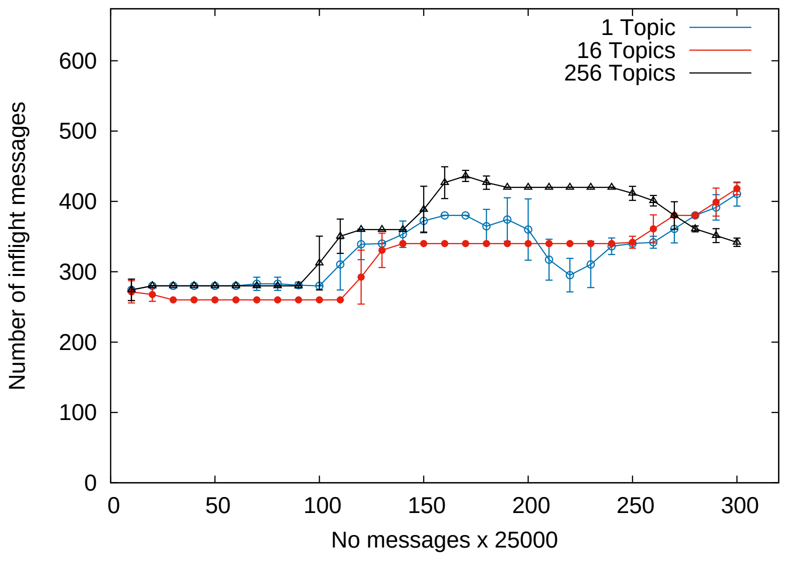


Fig. A.11 Number of in flight messages; Poisson Distribution

Latency

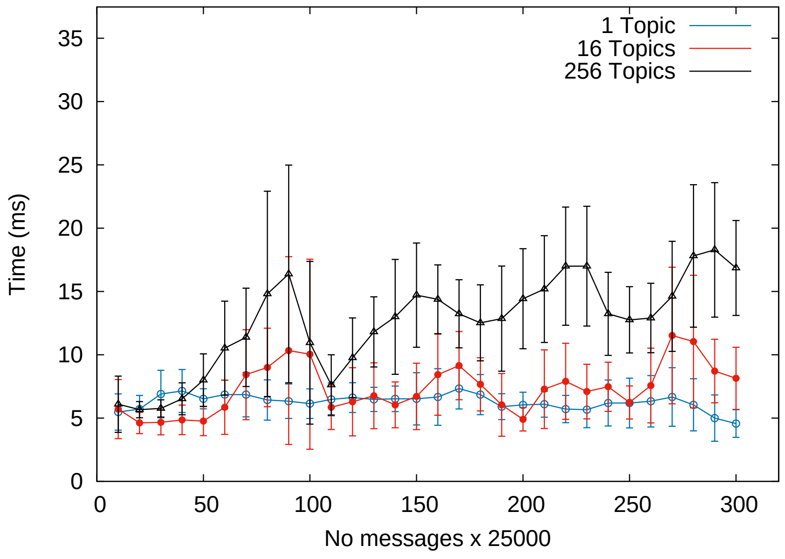


Fig. A.12 Latency; Normal Distribution

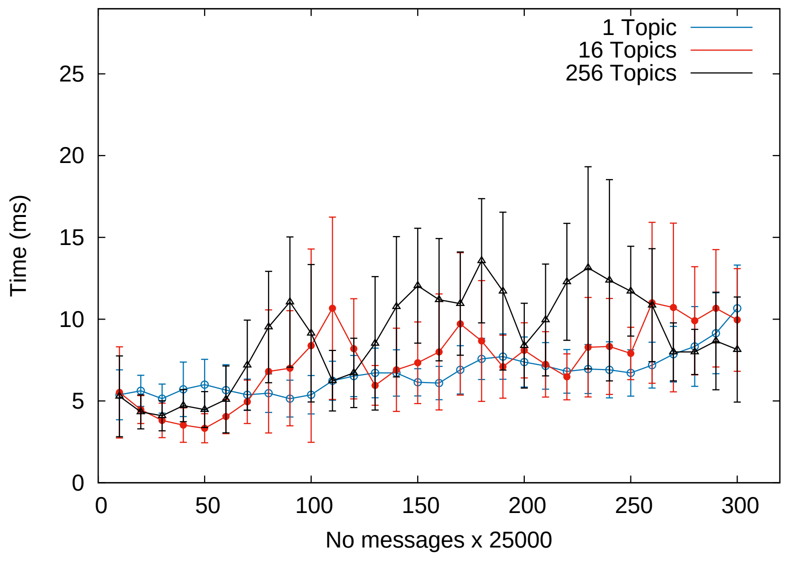


Fig. A.13 Latency; Poisson Distribution

Throughput

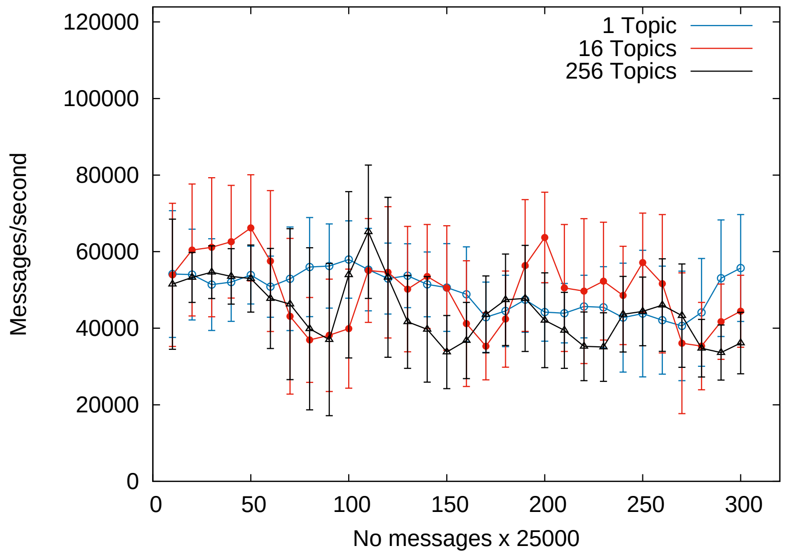


Fig. A.14 Throughput; Normal Distribution

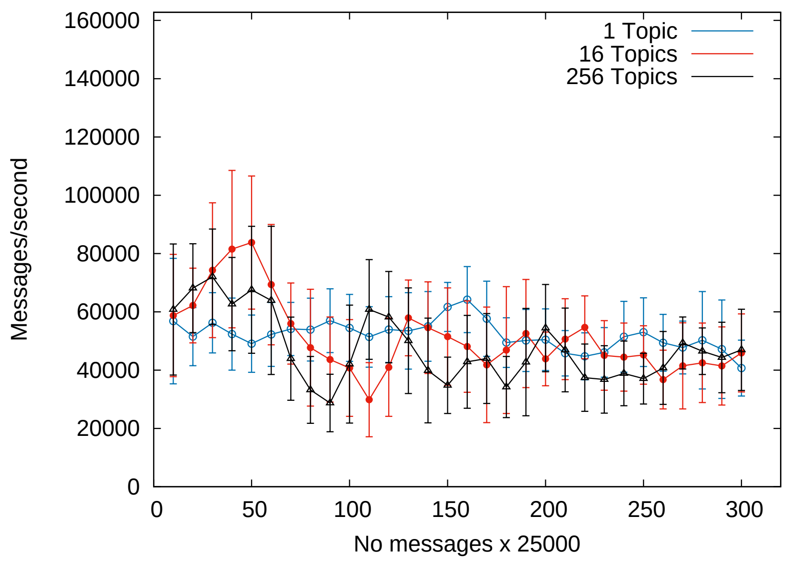


Fig. A.15 Throughput; Poisson Distribution

Little's Law

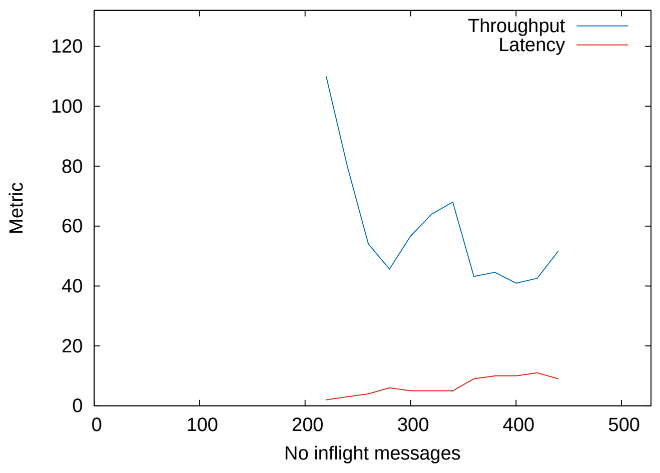


Fig. A.16 Relation between Throughput and Latency; Uniform Distribution; 256 topics

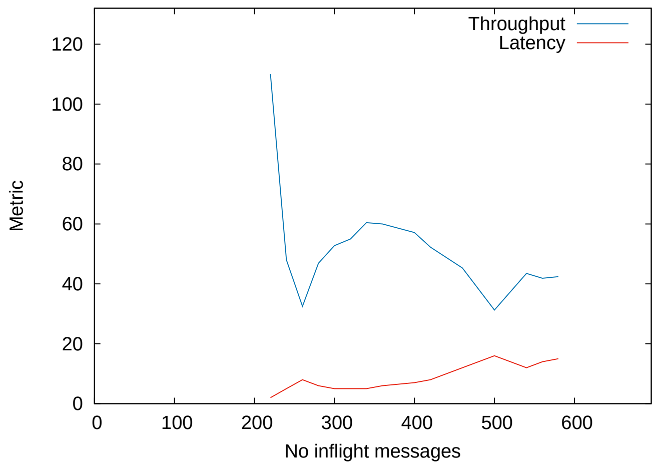


Fig. A.17 Relation between Throughput and Latency; Normal Distribution; 256 topics

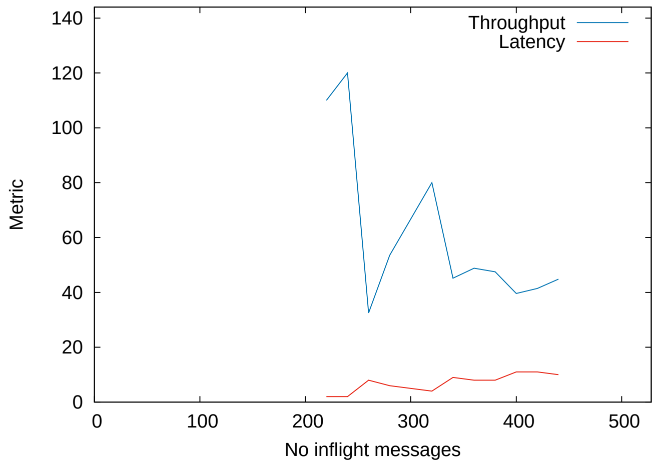


Fig. A.18 Relation between Throughput and Latency; Poisson Distribution; 256 topics

A.2 Kafka Log Storage

A.2.1 The Last Three Points Algorithm

In flight messages

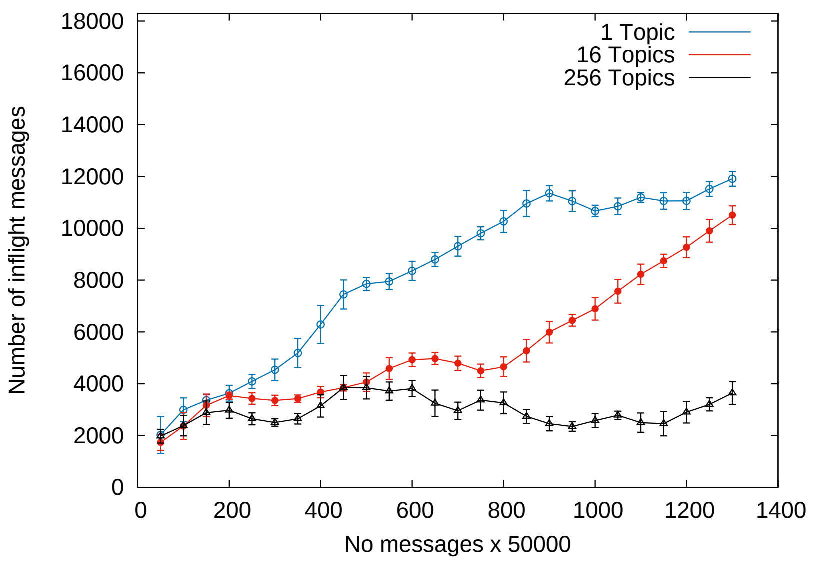


Fig. A.19 Number of in flight messages; Normal Distribution

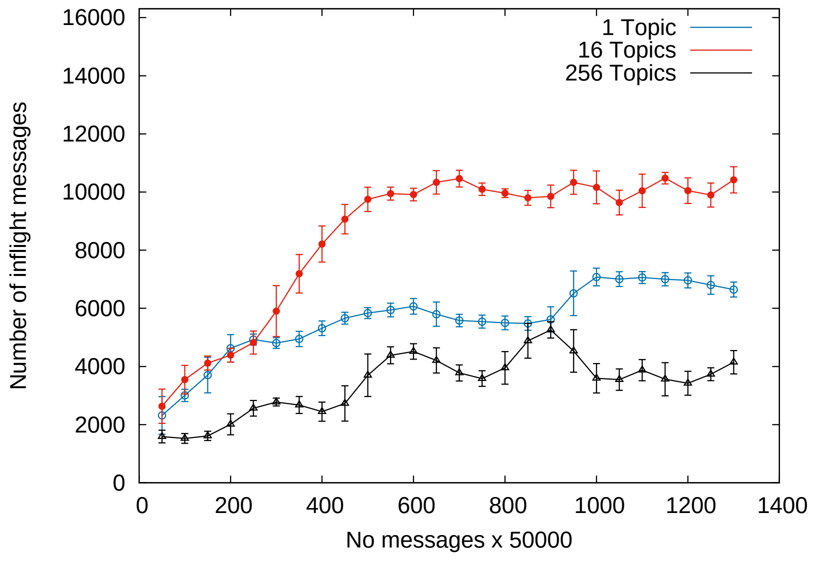


Fig. A.20 Number of in flight messages; Poisson Distribution

Latency

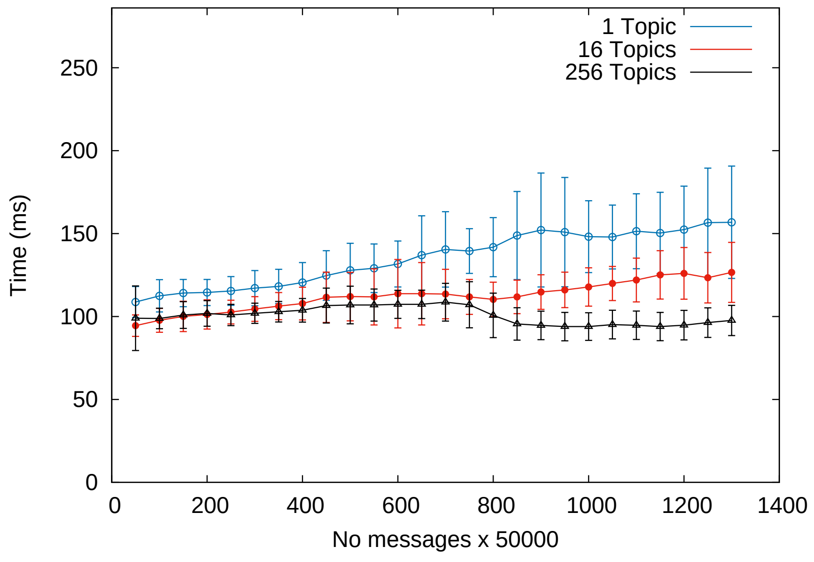


Fig. A.21 Latency; Normal Distribution

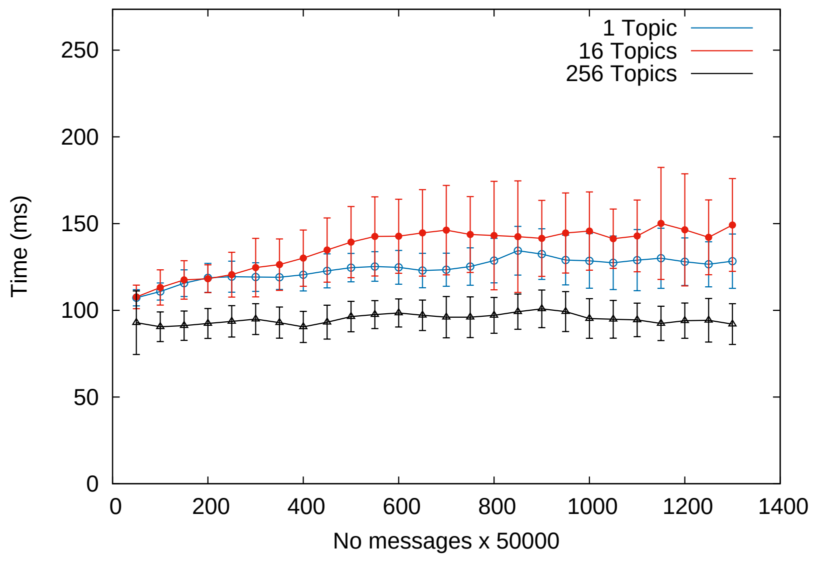


Fig. A.22 Latency; Poisson Distribution

Throughput

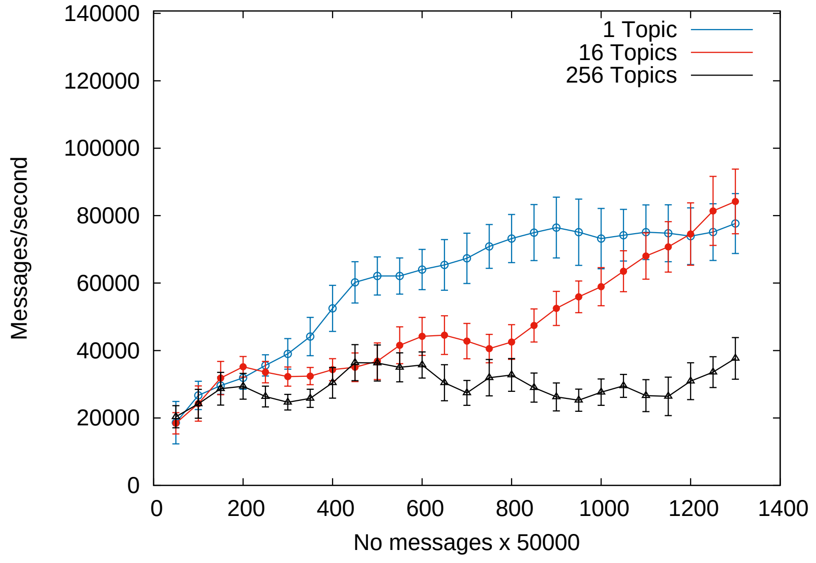


Fig. A.23 Throughput; Normal Distribution

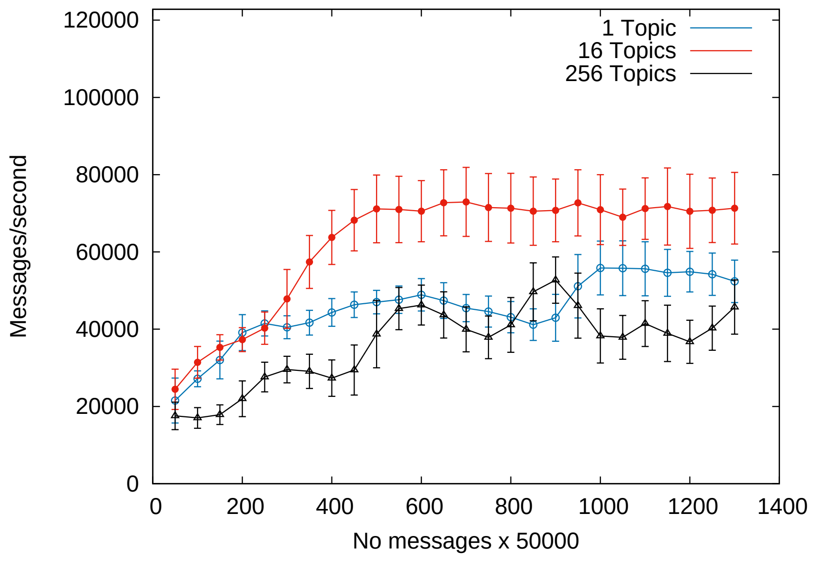


Fig. A.24 Throughput; Poisson Distribution

Little's Law

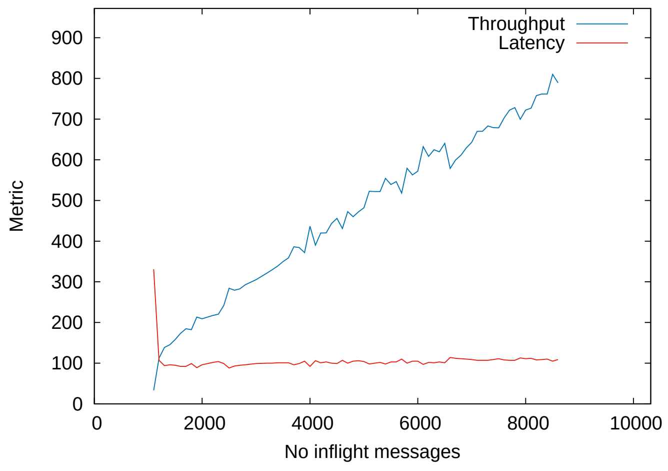


Fig. A.25 Relation between Throughput and Latency; Uniform Distribution; 256 topics

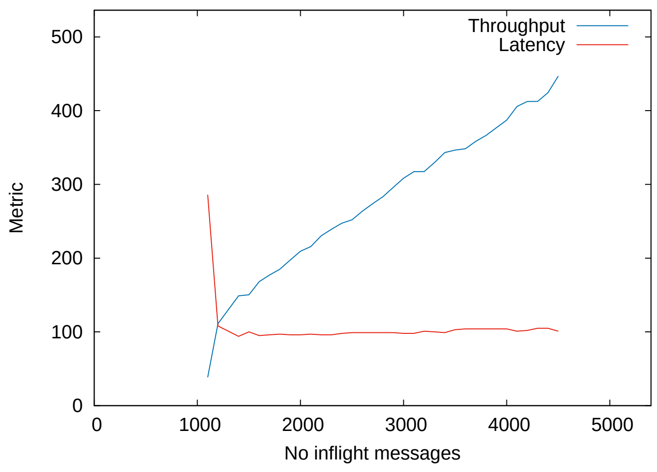


Fig. A.26 Relation between Throughput and Latency; Normal Distribution; 256 topics

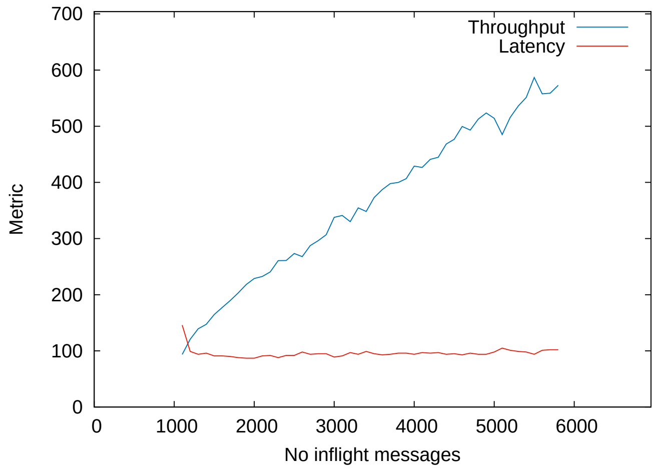


Fig. A.27 Relation between Throughput and Latency; Poisson Distribution; 256 topics

A.2.2 Adaptive Algorithm

In flight messages

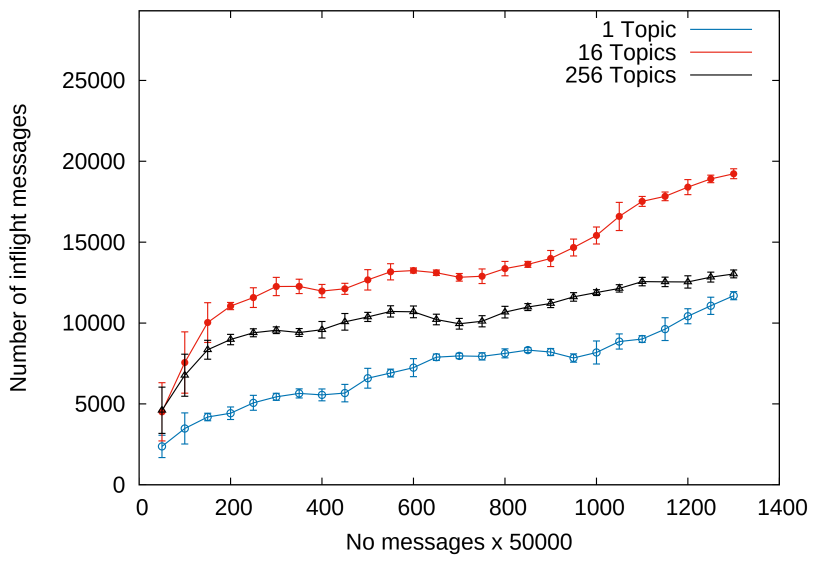


Fig. A.28 Number of in flight messages; Normal Distribution

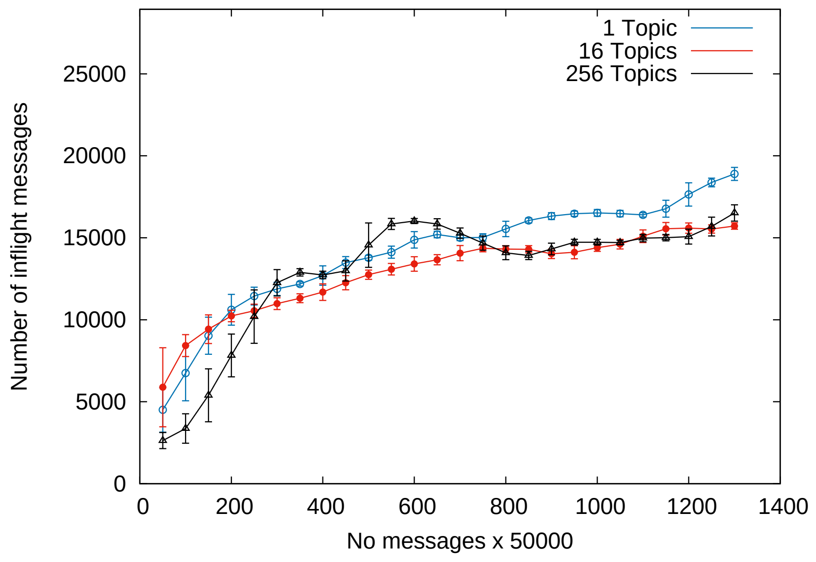


Fig. A.29 Number of in flight messages; Poisson Distribution

Latency

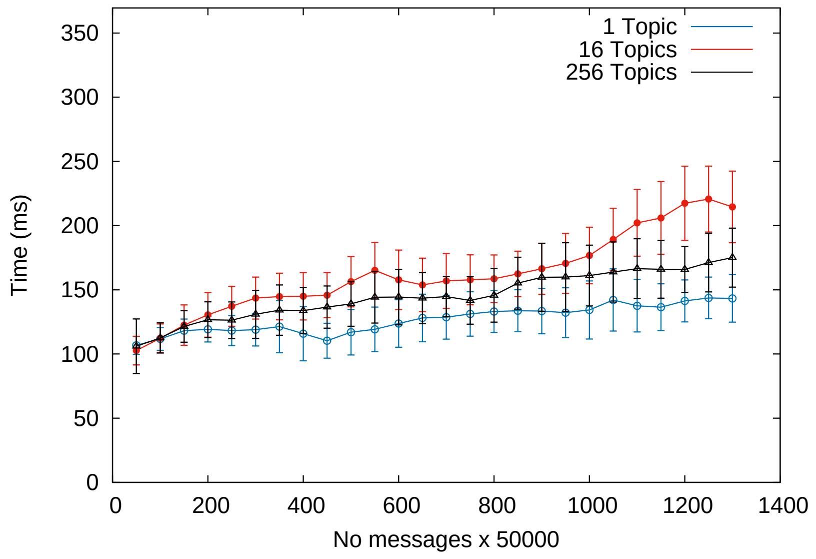


Fig. A.30 Latency; Normal Distribution

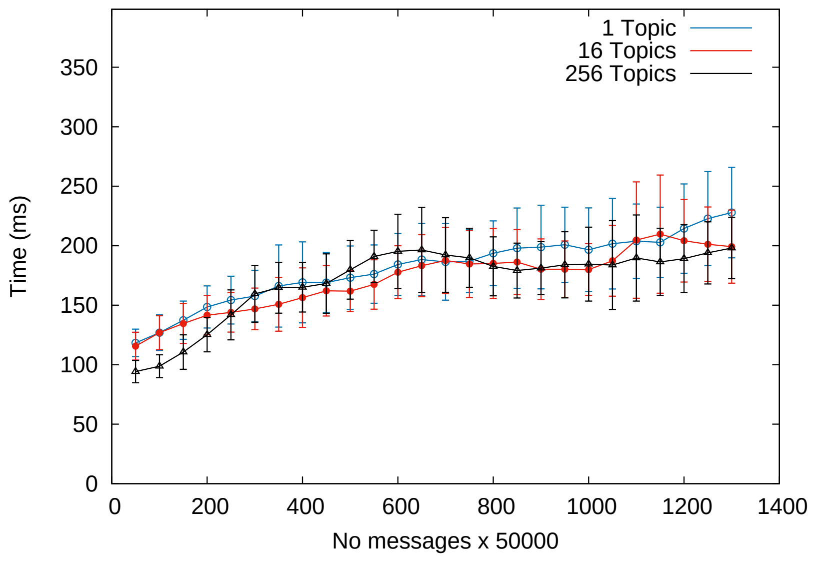


Fig. A.31 Latency; Poisson Distribution

Throughput

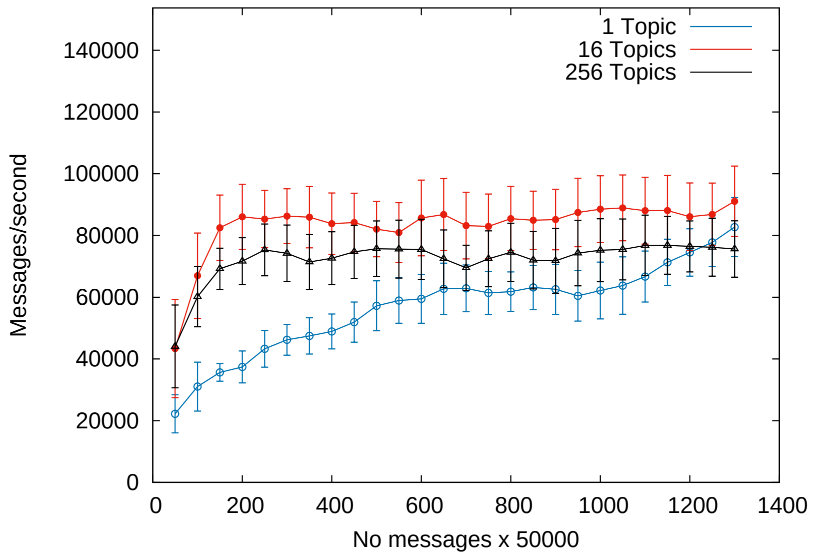


Fig. A.32 Throughput; Normal Distribution

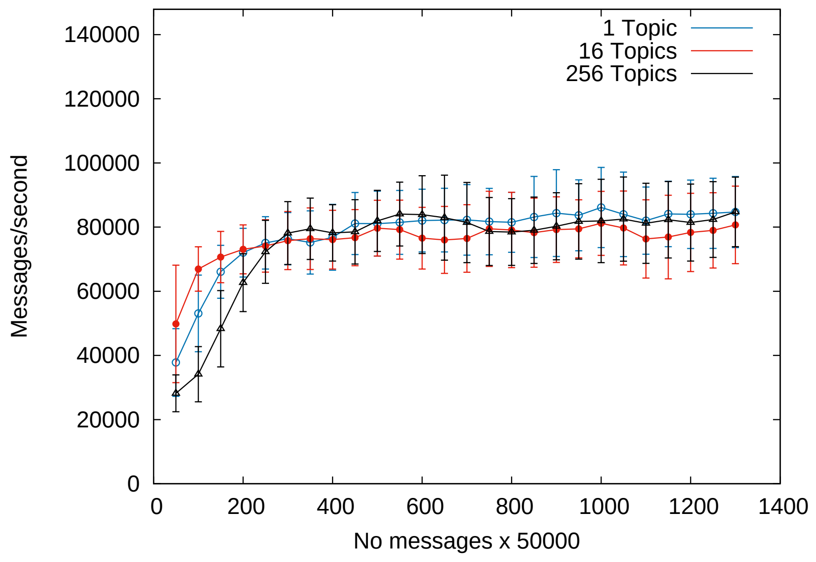


Fig. A.33 Throughput; Poisson Distribution

Little's Law

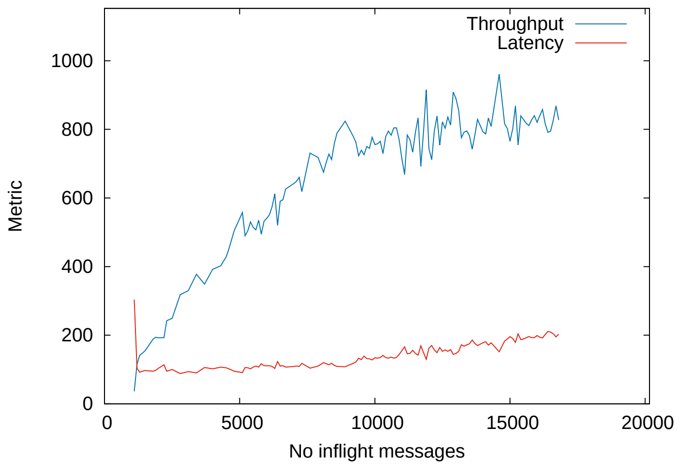


Fig. A.34 Relation between Throughput and Latency; Uniform Distribution; 256 topics

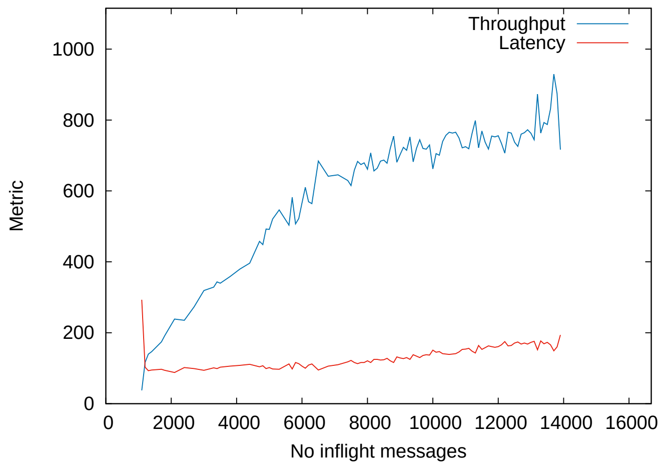


Fig. A.35 Relation between Throughput and Latency; Normal Distribution; 256 topics

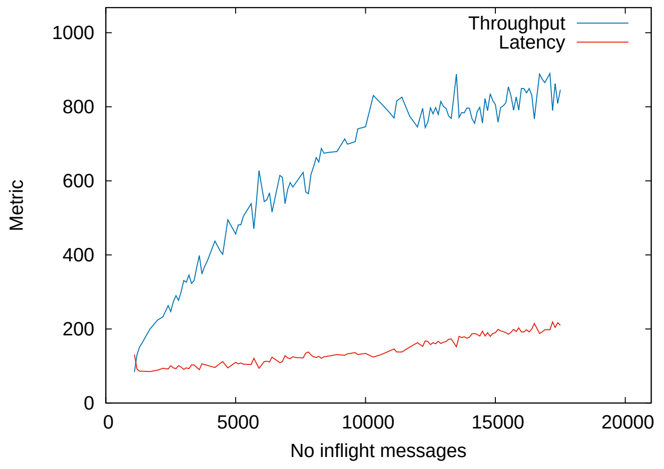


Fig. A.36 Relation between Throughput and Latency; Poisson Distribution; 256 topics