Ivar Sørbø

Scaling a Functional Measurement-Stream Transformation System using Pub/Sub Services

Master's thesis in Computer Science

Supervisor: Svein-Olaf Hvasshovd, IDI
Co-supervisor: Tommy Jakobsen, Kongsberg Digital AS

June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

KONGSBERG

Ivar Sørbø

Scaling a Functional Measurement-Stream Transformation System using Pub/Sub Services

Master's thesis in Computer Science
Supervisor: Svein-Olaf Hvasshovd, IDI
Co-supervisor: Tommy Jakobsen, Kongsberg Digital AS
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Norwegian University of
Science and Technology

Abstract

As more and more industrial sensors get connected to the Internet, high demands are put on the systems handling these data. Clients want to monitor and analyze the data from their sensors, both as real-time measurement streams and as historical time series. Kongsberg Digital has developed one such system which also allows the clients' queries to seamlessly transition from historical to real-time data. One of the challenges with this system, however, is that the current solution doesn't scale with the number of clients.

In this thesis, an architecture that increases the scalability of such a system is designed by using publish-subscribe systems to fan-out results that are shared among multiple clients while still allowing for a seamless transition from historical to real-time data. The architecture is then implemented and tested in a proof-of-concept system to verify the result.

Sammendrag

Nå som flere og flere industrielle sensorer kobles til Internet, stilles det høye krav til systemene som håndterer disse dataene. Klientene ønsker å overvåke og analysere måledataene sine, både i sanntid og som historiske tidsseriedata. Kongsberg Digital har utviklet et system for dette formålet. Dette systemet gjør det også mulig for klientene å sende spørringer som sømløst går i fra å levere historiske data til å levere sanntidsdata. En av utfordringene med dette systemet er at det ikke skalerer med antall klienter som bruker det.

I denne oppgaven er det designet en system arkitektur som forbedrer skalerbarheten til et slikt system. Dette oppnås ved å bruke et «publish-subscribe-system»til å spre ut samme resultat til alle klientene som har spørringer som resulterer i samme samtids-resultat. Samtidig beholdes den sømløse overgangen fra historiske data til sanntidsdata. Akritekturen er deretter implementert og testet i et test-system for å verifisere at det fungerer.

Preface

This masters thesis is the final project of my two-year master degree in Computer Science at the Norwegian University of Science and Technology (NTNU). The project is a cooperation between the Department of Computer Science of NTNU and Kongsberg Digital AS.

I would like to thank my supervisors Professor Svein-Olaf Hvasshovd from NTNU and Tommy Jakobsen from Kongsberg Digital for valuable inputs, feedback, and guidance throughout the semester.

Ivar Sørbø

Trondheim, May 2019.

Table of Contents

Abstract **i**

Sammendrag **ii**

Preface **iii**

Table of Contents **vi**

List of Tables **vii**

List of Figures **x**

Abbreviations **xi**

1 Introduction **1**
1.1 Objective 2
1.2 Limitations 2
1.3 Thesis Structure 2

2 Background and Related Work **3**
2.1 Galore 3
2.2 Multiple Query Optimization 4
2.3 The Lambda Architecture 5
2.4 Pub/Sub Services 6
2.4.1 Apache Kafka 7
2.4.2 NATS 7
2.4.3 RabbitMQ 8
2.5 Conclusion 8

3 Design Choices **9**
3.1 The Problem 9
3.2 Transition 9
3.2.1 The Buffer-based Transition Alternative 9
3.2.2 The Log-based Pub/Sub Alternative 9
3.2.3 Server-side vs Client-side Transition 10

3.3 Grouping Real-Time Streams 10
3.3.1 The Merging Alternative 10
3.3.2 The Dedicated Real-Time Publisher Alternative 13
3.4 Comparison and Chosen Approach 14
3.5 Selecting Pub/Sub System 15
3.6 Conclusion 15

4 System Architecture **17**
4.1 Component Overview 17
4.1.1 Information Flow 18
4.1.2 Queries and Messages 18
4.2 Server-side 19
4.2.1 The gRPC Server 19
4.2.2 Simulator 20
4.2.3 Making the Server-side Distributed 20
4.3 Client-side 21
4.3.1 Client Application 22
4.3.2 Client Library 22

5 Testing and Results **25**
5.1 Test Environment 25
5.1.1 System Settings 26
5.1.2 Performance Measures 26
5.2 Testing Different Usage Patterns 26
5.2.1 Environment Setup 27
5.3 Testing the Scalability 27
5.3.1 Environment Setup 27
5.4 Results 28
5.4.1 Different Usage Patterns 28
5.4.2 Scalability 29
5.5 Evaluation 30

6 Conclusion and Future Work **35**
6.1 Conclusion 35
6.2 Future Work 35

Bibliography **37**

A Sliding Buffer Data Structure **41**

B Raw Test Results **43**

List of Tables

3.1 Subscribers and corresponding topic-strings of a query group 12
3.2 Some example subscriber combinations and the corresponding topic-string used to publish to only them 13
3.3 Comparison of transition types 14
3.4 Comparison of transition place 14
3.5 Comparison of grouping methods 15

B.1 The test results for different usage patterns 43
B.2 CPU usage for the different usage patterns vs. the number of publishers. 44
B.3 Number of clients required to reach 40%, 50%, and 60% CPU load when the queries are distributed among multiple servers. 45

List of Figures

2.1 Streams of sensor data are feed to Galore from various industrial assets where it is made available for client applications for analysis and monitoring. 4
2.2 A simplified illustration of the lambda architecture where the incoming data is dispatched to two paths. 5
2.3 A publish-subscribe system where a purple and a green topic is published to a broker that routes the messages to the subscribers. 6
2.4 Overview of the main components of RabbitMQ and their interaction. 8

3.1 State 1: Client 1 and client 2 are interested in the same query. Client 1 is receiving real-time data while client 2 is still receiving historical data. 11
3.2 State 2: Client 2 has now caught up with client 1. The two publishers are now doing the same work. 11
3.3 State 3: The two streams/topics are merged. A single publisher now publishes to both clients. 11
3.4 State 1: Both clients are interested in the same query. Client 1 is receiving real-time data while client 2 is still consuming historical data. The historical data is sent by a temporary publisher. 13
3.5 State 2: Both clients consume real-time data from the same publisher. The temporary historical publisher has been terminated. 14

4.1 Information flow between the main components 17
4.2 A more detailed illustration of the server-side. 19
4.3 Using a master gRPC server to distribute the queries onto multiple servers. . . . 21
4.4 A more detailed illustration of the client-side 21
4.5 Transition phase 1: Forwarding historical stream. 22
4.6 Transition phase 2: Forwarding from the buffer. 23
4.7 Transition phase 3: Forwarding from the real-time stream. 23

5.1 Environment setup for testing different usage patterns on a single server. . . . 27
5.2 Environment setup for testing how the system scales. The master server and NATS were run on the same VM. 28
5.3 Result of testing different usage patterns. The blue, red, yellow, and green line shows how the CPU load increases as the number of clients increase when the average number of clients per publisher is one, two, four, and ten. 29

5.4 Results from testing the scalability. The plot shows how many unique topics it takes to reach an average CPU load of 40%, 50%, and 60% for the servers, illustrated by the green, blue, and red dots. Linear lines are drawn for comparison. 30
5.5 Results from testing different usage patterns, but normalized to the number of unique real-time result streams. A detailed table of the results can be found in table B.2 in the appendix. 31

A.1 A sliding buffer consisting of 6 buckets of 10 seconds each. The leftmost bucket has just been removed. 41

Abbreviations

API	=	Application Programming Interface
EOF	=	End-of-File
FIFO	=	First in, First out
IIoT	=	Industrial Internet of Things
MQO	=	Multiple Query Optimization
OLAP	=	On-Line Analytic Processing
Pub/sub	=	Publish-subscribe
RPC	=	Remote Procedure Call
VM	=	Virtual Machine
XML	=	Extensible Markup Language

Chapter 1

Introduction

This thesis is based on Kongsberg Digitals' Industrial Internet of Things (IIoT) sensor database Galore. Galore is a system that receives streams of measurement data from various sensors, stores it, and performs functional transformations based on client queries. The queries can involve both real-time and historical data.

A typical approach to building systems that utilize both real-time and historical time series is to dispatch the data down two different paths, one for real-time data and one for historical data. This is known as a **lambda architecture**. One of the downsides of having two different paths is that the clients are usually provided with two completely different Application Programming Interfaces (APIs) and query languages. This makes it difficult to develop user applications that integrate both historical and real-time data. Integration of the two paths is a problem often left for the client application to solve.

Galore has taken a slightly different approach where both the real-time and the historical data is available through the same subscription-based API. This allows the users to send queries that start with historical data and when it reached the end of that data, it continues with real-time data. One of the challenges of such queries is that it takes a bit of time to store the measurements to the database, causing a delay between the database and the real-time stream. In Galore, the transition from historical to real-time data is handled on the server-side. This simplifies the development of client-application. However, it does have some scalability issues.

The usage pattern of Galore is such that many clients send queries that result in the same real-time data stream. For example, two clients might send the same query at different points in time or by using different query-formulations that lead to the same processes being applied. With the current solution, each of these queries needs its own publisher which gathers the required data from the database and/or real-time streams, processes it, and publishes the result to the client. Contrary to a query for a normal database, a stream query can keep running indefinitely. Thus, a typical query in Galore spends most of its lifetime consuming real-time data. This is not a big problem when dealing with a few clients, but as the client count increases the number of duplicate publishers also increases.

A publish-subscribe (pub/sub) system can easily be used to scale the real-time part to handle more clients as it separates the publishers from the clients (subscribers) by categorizing the messages instead of sending all of them one by one. This allows a single publisher to send the same data stream to multiple clients. A downside of this is that it is less trivial to integrate

real-time and historical data processing.

This leads to the research question of this thesis:

“How can a pub/sub service improve the scalability of Galore to support more clients while still handling the transition from historical to real-time data streams assuming most of the queries result in equal real-time streams?”

The thesis is aimed to fit any system that uses a similar architecture as Galore and therefore does not focus too much on the specifics of Galore.

1.1 Objective

The objective of this thesis is to develop a simplified model of Galore and attempt to scale it for an increased number of clients by using a scalable pub/sub service. It is assumed that most of the requests have identical real-time results.

The system should only need one producer for each "group" of queries that results in the same real-time result. At the same time, the task of transitioning from historical to real-time data should not be left for the client application to solve.

1.2 Limitations

One of the challenges of scaling this system is to figure out which queries result in the same real-time result. Galore uses a rich query language where the same real-time result can be achieved from multiple different query formulations. Analyzing the queries and figuring out which ones results in the same real-time streams is not part of this thesis.

There exist lots of different pub/sub systems on the market today, each with their pros and cons. This thesis does not aim to analyze and compare various pub/sub systems to find the most suited one for this specific purpose. It explores techniques suitable for most topic-based pub/sub systems.

Sharing resources for the historical part of a query is not a part of this thesis The historical portion of a query must be sent by a dedicated publisher. It is therefore assumed that the queries will spend the majority of their lifetime consuming real-time data.

1.3 Thesis Structure

The rest of the thesis is structured as follows. Chapter 2 presents some background theory and previous work related to this thesis. In chapter 3, the problem is analyzed and different approaches to solving it are presented and discussed. Chapter 4 describes the suggested solution and how it is implemented in a proof-of-concept system. Chapter 5 describes and evaluates some testing done on the system as well as their results. Finally, chapter 6 concludes the thesis and suggests some future work.

Chapter 2

Background and Related Work

This chapter introduces research and technologies related to this thesis. The thesis does not build on one specific field of research, as the use case for this problem is quite unique, but borrows ideas from a combination of different fields.

Section 2.1 introduces Galore, the system behind the problem that this thesis explores. Section 2.2 introduces **Multiple-Query Optimization**, a research field focused on saving resources for similar queries, both for traditional database queries and for stream queries. Section 2.3 introduces the **lambda architecture**, an architecture suited for systems that utilize both historical and real-time data. Section 2.4 describes the publish/subscribe messaging pattern along with some systems implementing the pattern. Finally, section 2.5 concludes the chapter.

2.1 Galore

Galore is an IIoT measurement stream transformation and storage system built by Kongsberg Digital AS [1]. Sensor data is constantly streamed to Galore from various industrial assets where it is stored, processed and made available to clients as illustrated in figure 2.1. The input can also contain external data such as weather forecasts and electricity prices.

Galore allows the clients to query the data from their assets both in real-time and for historical data. It is also possible for queries to start with historical data and, when it reaches the end, switch to using real-time data.

Galore's customers are typically owners of large industrial assets e.g., a wind farm, a fleet of vessels, oil rigs, etc. and the client-applications are typically programs that monitor and analyze the state of these assets. An example query for a wind farm could be to ask for the power output for all the wind turbines starting from 24 hours ago and continuing up to today and onwards.

Galore contains its own proprietary query language called TQL. The TQL has a set of operation keywords that can be chained into functional pipelines, for instance; aggregations, stream merges, normalization, fast Fourier transformations, and so on [2]. This allows the clients to build complex queries for all sorts of analysis and monitoring purposes.

Galore's customers often have multiple client-applications that monitor their data. Some monitor the same things, while some focus on different areas. Even if the applications monitor different things, they often have some overlapping queries i.e., queries that results in the same real-time stream. These overlapping queries are the focus of this thesis.

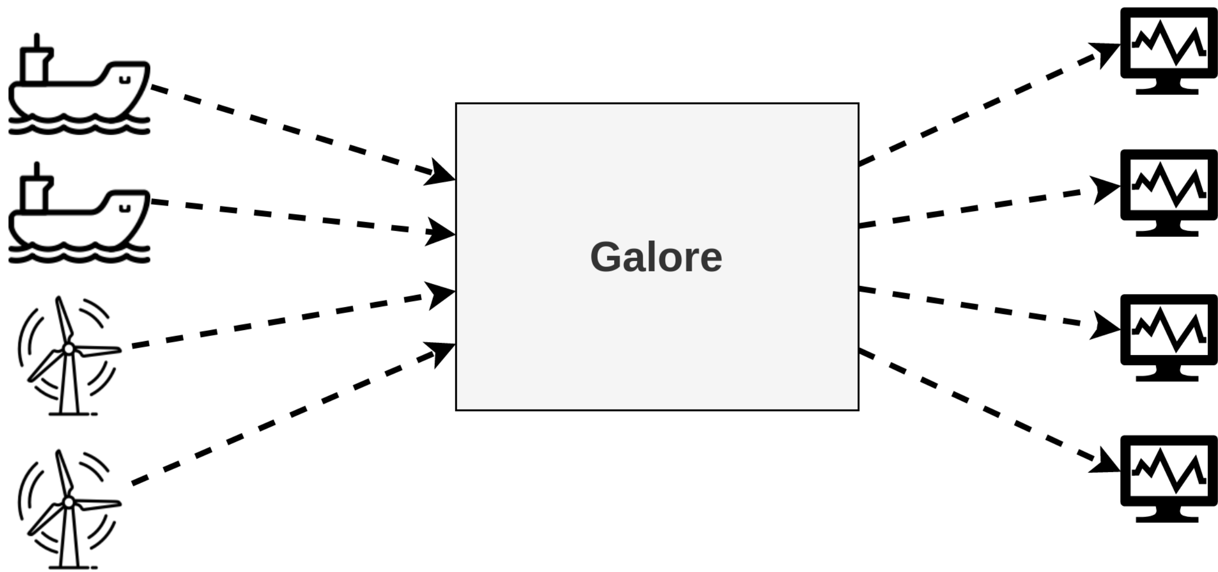


Figure 2.1: Streams of sensor data are feed to Galore from various industrial assets where it is made available for client applications for analysis and monitoring.

2.2 Multiple Query Optimization

Multiple Query Optimization (MQO) is the task of processing a batch of queries in the most efficient manner, both for traditional database queries and streaming queries [3].

Early work on MQO for traditional, static databases mostly focuses on finding the optimal evaluation plan for a small group of queries by locating and reusing computations of subexpressions shared between the queries [4], [5]. Optimizing a group of queries can lead to more efficient evaluation compared to individually optimizing each query but it comes with a higher computational cost.

Early MQO is not suited for streaming queries for several reasons. Firstly, it is only cost-efficient for a small group of queries. Secondly, some queries risk getting a slower response as the system waits for more queries before processing. Third, the methods are not designed for an environment where queries are frequently added and removed.

More recent work in the field contains techniques that are designed for, e.g., Extensible Markup Language (XML) query processing [6], materialized view maintenance [7], On-Line Analytic Processing (OLAP) [8], and stream query processing [9] [10]. MQO techniques for stream query processing are the sub-field most relevant for this thesis. For example, Krishnamurthy et al. [9] designed a resource sharing technique designed for streaming queries that vary both in their selection predicates and periodic windows. This method, unlike the traditional ones, does not require any up-front computation.

It is possible to use some traditional techniques for the historical parts of the queries, but the queries of Galore usually spend most of their lifetime dealing with real-time data, so the potential gain is very limited compared to optimizing the real-time part.

This thesis takes a slightly different approach to MQO compared to most of the other studies. The query analysis part of the optimization tasked with dividing the queries into smaller parts and finding shared parts is heavily simplified. In this thesis, only queries that result in exactly the same real-time stream will share resources. Simultaneously, the queries optimized here have

the added difficulty that they may start with historical data and when it reaches the end, continue with real-time data.

2.3 The Lambda Architecture

The Lambda Architecture [11] is a generic, scalable, and fault-tolerant data processing architecture that combines both batch-oriented and stream-oriented processing methods in the same framework to handle massive quantities of data. In the lambda architecture, all the incoming data is dispatched into two paths, a hot path, and a cold path. The hot path is a real-time stream processing path whereas the cold path is a batch-oriented processing path that includes permanent storage of the messages in e.g., a time series database. An illustration of a simplified lambda architecture is shown in figure 2.2.

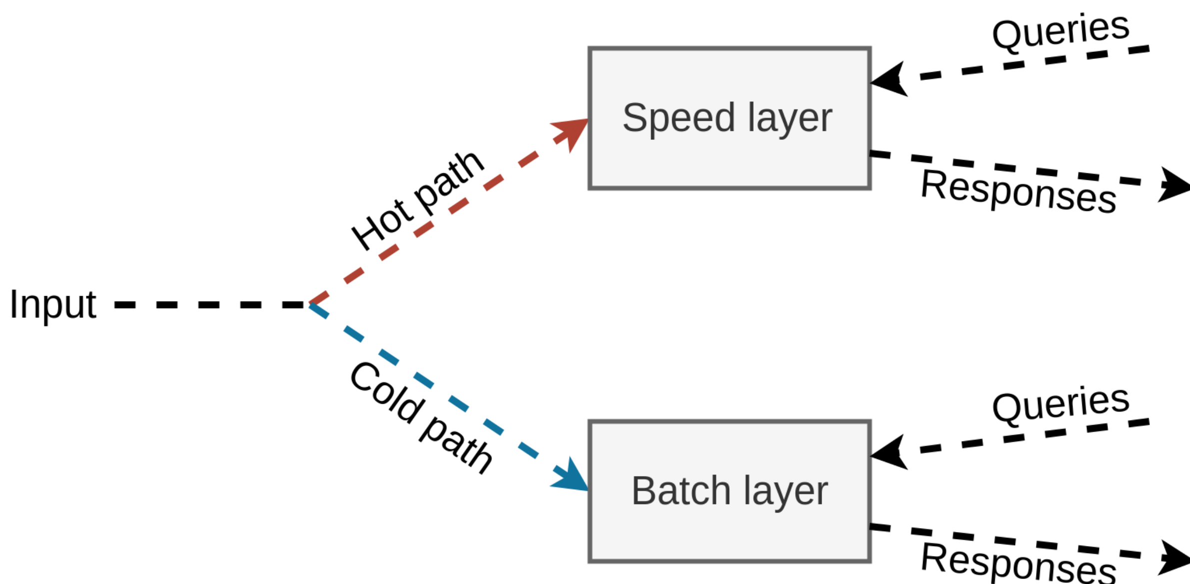


Figure 2.2: A simplified illustration of the lambda architecture where the incoming data is dispatched to two paths.

The lambda architecture is made up of three layers; batch layer, serving layer, and speed layer. The batch layer manages the master dataset, an immutable, append-only collection of data. The serving layer acts as an interface to the queries, both for the batch layer and for the speed layer. The speed layer compensates for the slow batch layer, often by approximating the results. This allows queries to get both precise results from the batch layer and quick results from the speed layer. The lambda architecture is by itself just a paradigm and can be implemented with different types of software to fit many different use cases, not just time-series data [12].

The lambda architecture is relevant for this thesis as it has some similar features to the underlying architecture of Galore and as well as being a popular choice for other systems that uses both real-time and historical data.

2.4 Pub/Sub Services

Publish-subscribe [13] is a messaging pattern consisting of publishers, subscribers, and typically a broker in the middle. The publisher publishes a message to a broker, and the broker forwards the message to all the subscribers who are interested in the message. There are two main filtering methods used to determine if a subscriber is interested in a message; content-based [14] and topic-based [15]. In content-based filtering, the internal properties of the messages are used to classify them. In topic-based filtering, the publishers assign a topic or a subject to each message which is used to classify them. In this thesis, only pub/sub systems with topic-based filtering are considered.

The subscribers sends subscription requests to the broker indicating which topics they are interested in. This topic abstraction creates a layer of separation between the publishers and subscribers.

Figure 2.3 shows a typical pub/sub system. Two publishers publish messages to different topics, a purple topic, and a green topic. The broker then routes these messages to the subscribers. The publishers and the subscribers do not need to know about one another as the topic abstraction is used to route the messages.

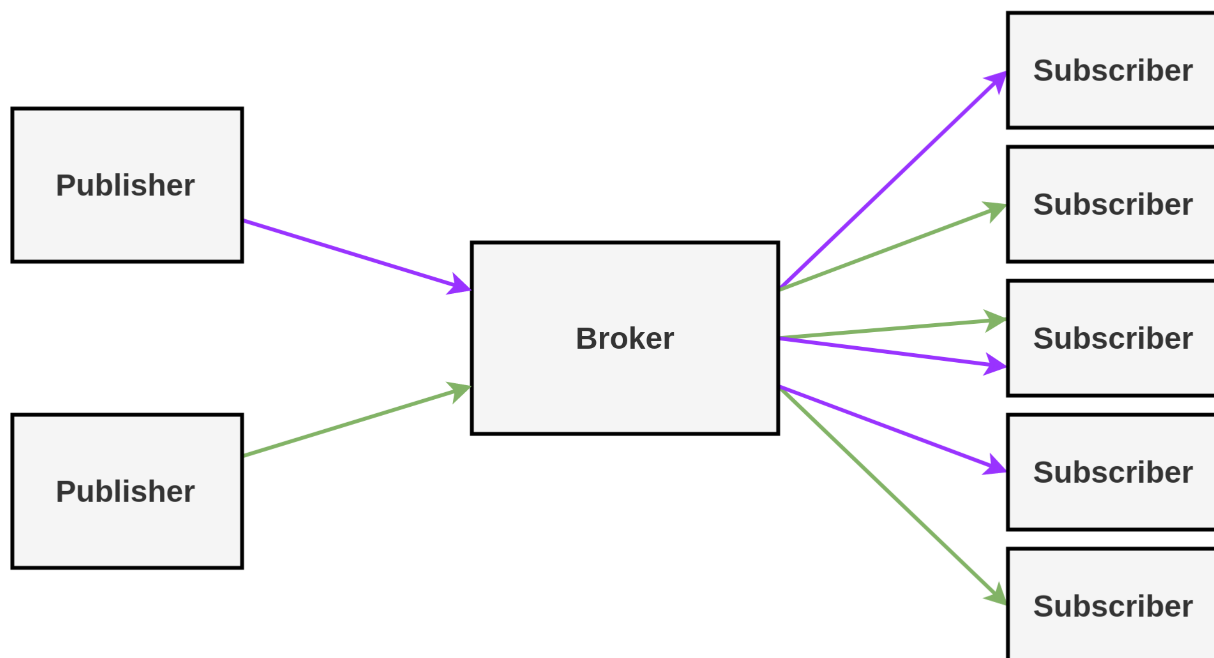


Figure 2.3: A publish-subscribe system where a purple and a green topic is published to a broker that routes the messages to the subscribers.

The topics used in a pub/sub system usually form a hierarchical structure [16]. For example, a topic used in a system monitoring wind farms could be:
`farm3.windturbine21.nacelle.windspeed`
indicating that the message contains the wind-speed measured at the nacelle of wind turbine number 21 at wind farm number 3. A "." is used as a delimiter between the different levels of the hierarchy. The string of characters used as an identifier within a specific level in the hierarchy (e.g. "windturbine21") is called a token.

Many pub/sub services also allow for wildcard characters in the topic. Wildcard characters allow subscribers to listen to multiple different topics with the same subscription. For example,

the pub/sub system NATS allows two types of wildcard characters; "*" and ">" [17]. The "*" will match a single token and ">" will match one or more tokens. A subscription to the topic:

`farm3.*.nacelle.windspeed`

means that the subscriber will receive wind-speed measurements from the nacelle of all the wind turbines of wind farm number 3 and a subscription to the topic

`farm3.>`

will receive all messages from wind farm number 3.

A pub/sub service is relevant for this thesis due to its ability to easily send a message to many subscribers while keeping a layer of separation between the source of a message and its recipients.

There are lots of different messaging systems that support different flavors of the pub/sub pattern. Three popular choices today are Apache Kafka [18], RabbitMQ [19], and NATS [20]. Each of these has its own traits and use cases.

2.4.1 Apache Kafka

Apache Kafka [18] is a distributed streaming platform that, along with processing and storage of streams, allows clients to publish and subscribe to streams of records (Kafka's equivalent of a message). Kafka uses write-ahead logging to write all the records in a redundant and fault tolerant way to disk as they arrive. The records contain a key, a value, and a timestamp.

Kafka is built with a focus on clustering. It is meant to be run on a cluster of computers to balance the load and to offer redundant storage of the log. Kafka is horizontally scalable and can run on anything from a single server to huge clusters that span multiple data-centers.

Kafka's use of a log structure to represent the stream leads to some interesting characteristics compared to other pub/sub services. Typically, the broker in a pub/sub system would control the cursor (i.e., the read-position in the stream) but in Kafka, this is the consumers' responsibility. This means that the consumer can start reading from any point in the stream, it can read at its own pace, and it can replay parts of the stream.

The main strengths of Kafka compared to a lot of other pub/sub services is its support for very high producer throughput, durability, fault tolerance, and support for both fast and slow consumers. The high throughput is achieved by dealing with messages in batches, but this also means that latency tends to be higher than other pub/sub services.

2.4.2 NATS

NATS [20] is a simple, high performance, and highly scalable messaging system supporting three different messaging patterns; pub/sub, load balanced queue, and request/reply. The load balanced queue is an extension to the "pure" pub/sub pattern which allows subscribers to be grouped and, a message sent to a group is only received by one of the subscribers. This is useful in e.g., load balancing and auto-scaling. The request/reply pattern allows for both one-to-one and one-to-many messaging where the messages contain a reply-topic for the subscriber(s) to send a reply back to the publisher.

The pub/sub messaging pattern of NATS is, unlike Kafka, a fire-and-forget system. This means that the broker controls the cursor of a stream and a subscriber needs to be actively

listening to the topic to receive a message.

Out of the three systems introduced here, NATS is the one with the lowest latency, but also the one with the lowest delivery guarantees. NATS only supports at-most-once delivery as it delivers messages immediately to the subscribers but does not persist them.

2.4.3 RabbitMQ

RabbitMQ [19] is a lightweight open-source message broker written in Erlang. It supports multiple messaging protocols, including MQTT, AMQP, HTTP, and STOMP. RabbitMQ consists of publishers, consumers (subscribers), exchanges, and queues.

Where Kafka uses a log and NATS uses fire-and-forget to get the messages from the publishers to the subscribers, RabbitMQ uses queues. The use of queues allows the consumers to read from the queue at their own pace similar to Kafka, but the messages can only be read once. RabbitMQ also support some persistency guarantees, both on queues and messages.

Figure 2.4 illustrates the different components of RabbitMQ. A publisher sends a message to an exchange, the exchange routes the message to the correct queues, and the consumers read from the queues. The messages can be routed based on topic, the message header, or it can be routed directly (point-to-point). The queues are append-only, first-in, first-out (FIFO) queues, and the messages are removed once they are read.

A queue can belong to one or more consumers. If two consumers read from the same queue, they compete for the messages. Alternatively, with multiple consumers wanting to read the same messages, multiple queues can be created.

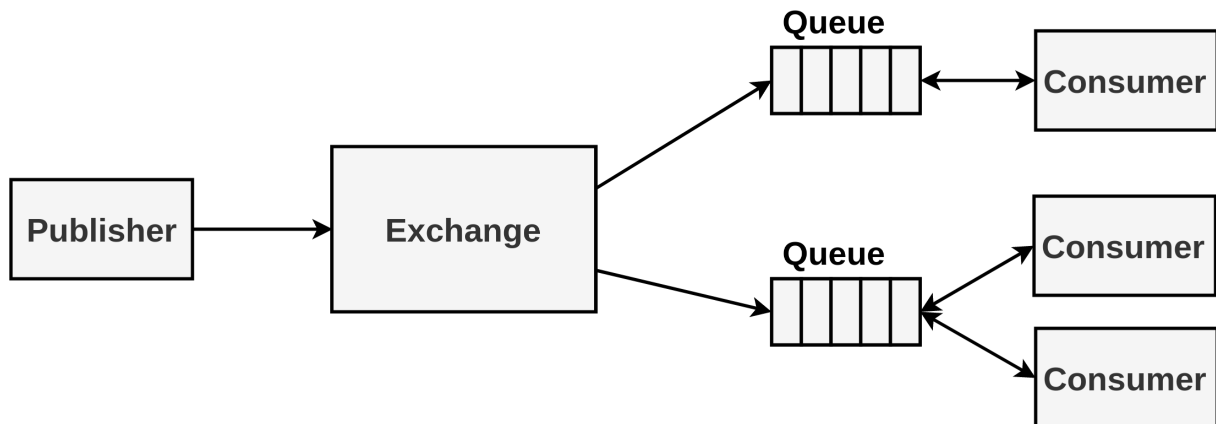


Figure 2.4: Overview of the main components of RabbitMQ and their interaction.

2.5 Conclusion

This chapter has introduced the main research fields and technologies that this thesis builds on. MQO share the same goal of saving resources by sharing results among multiple queries, the lambda architecture is a popular choice for systems that uses both historical and real-time data, and pub/sub services offers a good way to scale the real-time part of the system.

This thesis borrows ideas from all of these fields in the task of designing an architecture that fills the requirements presented in chapter 1.

Chapter 3

Design Choices

This chapter discusses the problem and different approaches to solving it. Section 3.1 analyzes the problem and breaks it into two main subproblems. Section 3.2 and 3.3 introduces different ways of solving these subproblems and discusses the pros and cons of the solutions. Section 3.4 summarizes the different approaches and discusses the chosen approach. Section 3.5 discusses the selection of a pub/sub system based on of the approach chosen in 3.4. Finally, section 3.6 concludes the chapter.

3.1 The Problem

As stated in section 1.1, the objective of this thesis is to develop a proof-of-concept system which only needs one producer for each query group (i.e., a group of queries that result in the same real-time stream) and that abstracts the task of transitioning from historical to real-time data streams away from the client application. The objective can be divided into two main components: grouping real-time streams and transitioning from historical to real-time data. The choices made in one of these subproblems affects, to some degree, the possibilities of the other, but they can still be analyzed separately.

3.2 Transition

Transitioning from historical to real-time data can be done in a couple of different ways, and it can be done either on the server-side or on the client-side.

3.2.1 The Buffer-based Transition Alternative

The simplest way to transition between the two streams is to use a buffer that stores the latest messages of the real-time stream so that it overlaps with the historical stream. This type of transition can be done on both the server-side and on the client-side.

3.2.2 The Log-based Pub/Sub Alternative

An alternative way for transitioning is to use a pub/sub system based on logging (f.ex. Apache Kafka) instead of the normal "fire-and-forget" type of pub/sub. In a log-based pub/sub system,

the stream is stored in a log data-structure and the client can control where in the log it should start reading from. This method eliminates the need for a buffer to store the real-time stream, but it limits the choice of pub/sub systems. This method implies that the transition happens on the server-side.

3.2.3 Server-side vs Client-side Transition

The transition from historical data to real-time data can either be done on the server-side or the client-side, each having its own traits.

The transitioning can be done on the client-side without relying on the client application to do it by using a client library. An argument for doing the transition on the client-side is that it makes the server-side simpler and allows for an architecture similar to the lambda architecture. A downside is that some of the messages will most likely be sent on both streams, thereby wasting bandwidth resources. Another thing to note is that only the buffer-based transition alternative is possible when using the client-side. If the buffer-based alternative is chosen, having to create a client-library is not necessarily a downside as the same functionality are required regardless of where the transition happens.

A benefit of doing the transition on the server-side is that it practically eliminates the need for a client library. Another benefit is that the publisher can ensure that the same message is not sent on two streams. It also allows for both the buffer-based and the log-based pub/sub transition alternative. One of the downsides of server-side transition is that the server becomes more involved in handling each query request which could make the system more difficult to fit into a distributed environment.

Choosing where to do the transition largely depends on what fits with the other parts of the solution. One side of the system is not necessarily better than the other.

3.3 Grouping Real-Time Streams

There are several ways of ensuring that there is only one producer per query group. The two main categories of options are to either merge two streams that have resulted in the same real-time stream or, to keep a dedicated real-time publisher for each query group and use temporary publishers for the historical data. Merging two streams fits best together with server-side transition while using dedicated real-time and historical publishers is better suited for client-side transition.

This section explores and compares different ways of approaching this problem.

3.3.1 The Merging Alternative

Perhaps the most intuitive solution is to keep the system the way it is now and, when two producers reach a point where they produce the same result, they are somehow merged into one producer. This can be done in multiple ways e.g., with a pub/sub system that allows external altering of the interest graph, it can be done with some cleverly designed topic structure based on wildcards, or it can be done with the help of the client. Merging streams is mainly suited for server-side transition. Figures 3.1, 3.2, and 3.3 show an example scenario of a topic merge divided into three different states.

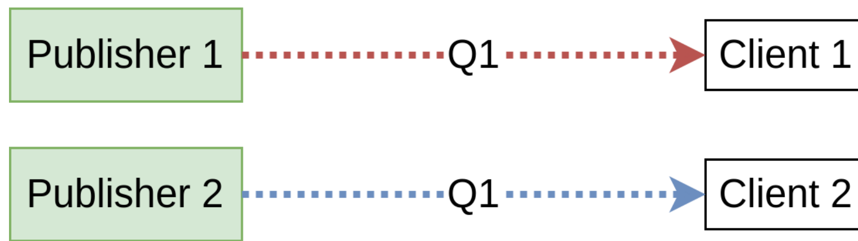


Figure 3.1: State 1: Client 1 and client 2 are interested in the same query. Client 1 is receiving real-time data while client 2 is still receiving historical data.

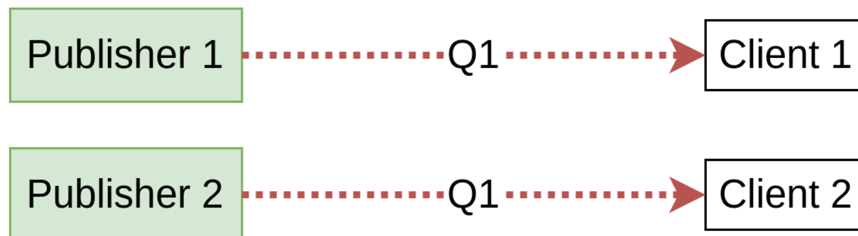


Figure 3.2: State 2: Client 2 has now caught up with client 1. The two publishers are now doing the same work.

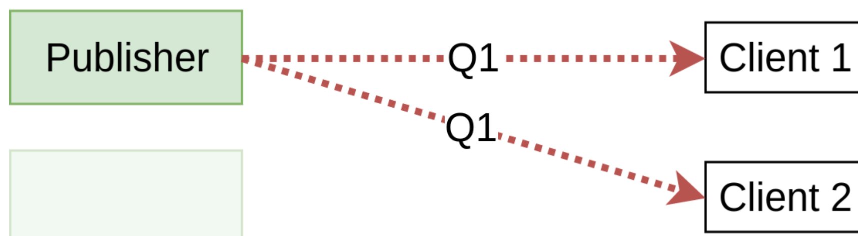


Figure 3.3: State 3: The two streams/topics are merged. A single publisher now publishes to both clients.

Based on Client Changing Topic

A simple way to merge two topics is to ask all the clients of one topic to subscribe to the other topic and to unsubscribe from their original topic. There are multiple ways to ensure that no messages are lost when a client changes the topic. E.g., an overlap between the subscribe and unsubscribe can ensure an at-least-once delivery guarantee.

One of the downsides of making the client change its topic is that it would require some functionality on the client-side to handle the switch. It would also complicate the server-side as the two publishers must first determine which one survives and which one dies before communicating this to the client(s).

Based on Altering the Pub/Sub Interest Graph

A pub/sub system that allows third parties to alter the interest graph (i.e. the graph connecting the subscribers with the topics that they are interested in), would allow the publishers to control the merge without the client noticing. This is an improvement compared to the previous method as it simplifies the client-side, but it does not solve the challenge of picking which publisher survives.

At the time of writing, a system with such functionality does not exist (at least not among the most common pub/sub systems).

Based on a Topic Structure with Wildcards

Some of the pub/sub systems allow the use of wildcard characters in the topic. Wildcard characters are usually meant for clients to decide which topics it is interested in and allows them to subscribe to multiple topics while using the same subscription. If, however, the topic string is decided by the server-side, it can also be used by the publishers to decide which clients a message is sent to through some cleverly designed topic structure.

Using the wildcard characters of NATS, an example topic structure that allows the publisher to decide which subscribers receive a message would be built as follows:

- Each query group decides on a shared token, ex. 'a'. This token should be as short as possible but has to be unique for that group.
- The system must also decide on a special token which is not to be used as a shared token for any of the query groups, ex. '0'.
- The topic for a query is then made up of a specific number of '*'-wildcards, followed by a single shared token for that particular query group, and finally a '>' wildcard. The first topic in a query group contains no '*'s, the next contains one '*', the third contains two '*'-wildcards, and so on.

Subscriber number:	Topic-string:
0	`a.>`
1	`*.a.>`
2	`*.*.a.>`
3	`*.*.*.a.>`
4	`*.*.*.*.a.>`

Table 3.1: Subscribers and corresponding topic-strings of a query group

Table 3.1 shows an example query group that allows the publisher to control which of the subscribers receives the message by changing which topic it publishes to. E.x. a message sent to the topic: `0.a.a` would be received by the subscribers at topics `*.a.>` and `*.*.a.>`, but not the others. Similarly, it is possible to use combinations of the special token ('0') and the shared token ('a') to decide any combinations of subscribers who is to receive a message.

Calculating which topic can be used to publish to a set of subscriber-topics is similar to doing a bitwise-OR operation on the topic-strings. For each token; if one of the topics contain the shared token at that position, use the shared token in the topic, otherwise, use '0'. Table 3.2 shows some example combinations of subscribers and which topic to publish to so that only the chosen subscribers will receive them.

The use case of this is when two publishers wants to merge into a single publisher, the surviving publisher could simply change which topic it publish to in order to include the new subscriber.

Only publish to:	Publish topic-string:
`a.>`	`a`
`*.*.*.a.>`	`0.0.0.a`
`a.>` and `*.a.>`	`a.a`
`*.a.>` and `*.*.*.a.>`	`0.a.0.a`
`a.>` and `*.a.>` and `*.*.*.a.>`	`a.a.0.a`
`a.>` and `*.a.>` and `*.*.a.>` and `*.*.*.a.>`	`a.a.a.a`

Table 3.2: Some example subscriber combinations and the corresponding topic-string used to publish to only them

This allows real-time streams to merge without the clients noticing. This method does, however, have a few downsides, mainly due to the pub/sub system no longer acting as a layer of abstraction between the publishers and the subscribers. The publishers need full control of which subscribers to send each message to and it has to recalculate the topic each time the list of subscribers changes. The system also needs to keep track of all the topics used in all the query groups, both for assigning new ones and to reclaim used ones when a client disconnects. Another downside is that the topic-string increases with the number of clients sharing a result stream. This method also limits the options of pub/sub systems as it must support wildcards and very long topic-strings.

3.3.2 The Dedicated Real-Time Publisher Alternative

Another alternative is to not merge equal streams but for each query-group to have a dedicated producer for real-time data. This would imply that a query that requires both historical and real-time data will need two streams and the transition happens either at the client-side or at an intermediate step at the server-side. The historical data needs to be sent on a temporary topic unique to a client, but the historical publisher can be terminated once the client transitions to real-time.

An advantage of this is that there is no need to determine which publisher survives as there is only one real-time topic per query group. This eliminates the need for topics to merge.

The downside of this is that the transition between historical and real-time cannot happen at the publisher but requires an intermediate step, either on the server-side or on the client-side. This limits the choice of transition methods to a buffer-based method.

Figure 3.4 and 3.5 shows an example scenario of this method.

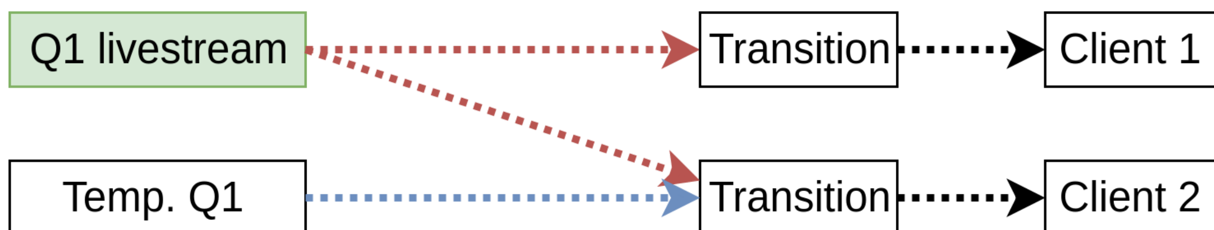


Figure 3.4: State 1: Both clients are interested in the same query. Client 1 is receiving real-time data while client 2 is still consuming historical data. The historical data is sent by a temporary publisher.

Figure 3.4 shows the initial status. Client 1 and client 2 are interested in the same query.

Client 1 is receiving real-time data from a dedicated source while client 2 is receiving historical data from a temporary source. Client 2 also knows about the dedicated real-time source of its query but does not need to subscribe to it until it is near the end of the historical stream.

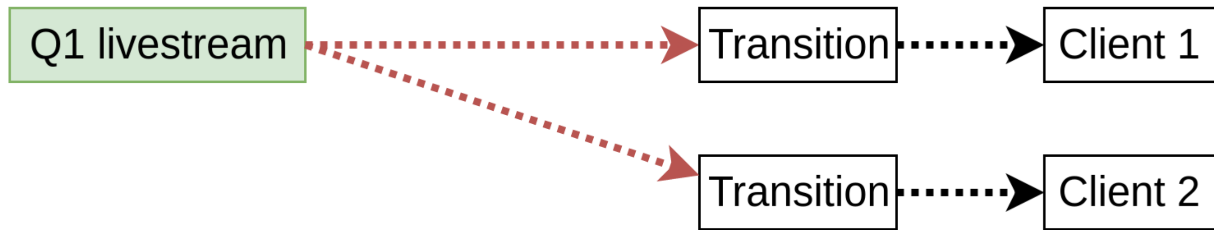


Figure 3.5: State 2: Both clients consume real-time data from the same publisher. The temporary historical publisher has been terminated.

Figure 3.5 shows the status when client 2 has reached real-time. The temporary source is no longer needed as client 2 is only interested in the real-time data. Both clients receive the real-time stream from the same source.

3.4 Comparison and Chosen Approach

As seen in the previous sections there are multiple approaches to the problem, both for transitioning and for grouping. Tables 3.3, 3.4, and 3.5 shows a quick comparison of the pros and cons of each of the methods.

Buffer-based:	**Log-based:**
Only client-side or intermediate server-side transition	Allows for both client-side and server-side transition
Available for any pub/sub system	Only available for log-based pub/sub systems

Table 3.3: Comparison of transition types

Server-side:	**Client-side:**
Simple client-side, no need for client-library	Requires a client library
The server is more involved in each query request	Better separation between server and client
Allows the transition to be built into the publishers	Allows for an architecture similar to the lambda architecture
Can be built to ensure no messages are sent twice	Requires a slight overlap of historical and real-time data when transitioning

Table 3.4: Comparison of transition place

This thesis aims to propose a general solution to the problem at hand without too much focus on a specific pub/sub system or on the specifics of Galore. It should also be easy to make the system distributed.

Merge:	Dedicated real-time:
Requires the transition to be built into the publishers	Allows both client-side and intermediate server-side transition
Server-side requires some extra complexity to facilitate the merging	Trivial to implement the server-side

Table 3.5: Comparison of grouping methods

The chosen transition type is a buffer-based approach as it puts much fewer constraints on the available pub/sub services. A dedicated real-time publisher is chosen as the grouping method and the transition is placed at the client side. All of this makes the server-side much simpler and less bound up by each query request. Keeping the transition separated from the publishers makes for simpler and more independent publishers. This makes the server-side better suited for a distributed environment.

It would also be possible to choose to do the transition at an intermediate step on the server-side but this would require the server-side to set up a transition-operation for each query request that require a transition. It would also lead to a higher bandwidth cost within the server-side. Although, if the bandwidth between server and client is a big concern it might be a better choice.

3.5 Selecting Pub/Sub System

Another big choice is to decide which pub/sub system to use. As stated in chapter 2.4, there are a lot of different pub/sub systems on the market today, each with its own traits. The choice of pub/sub system heavily depends on the usage patterns and specific needs of the system. The aim of this thesis is not to benchmark and compare a lot of different pub/sub systems but to propose a solution that fits most pub/sub systems. The only basic requirements for the pub/sub system is that it is scalable and have a relatively fast throughput.

NATS is selected for the implementation in this thesis as it is the simplest pub/sub system of the ones studied here. A solution that works well with a fire-and-forget system like NATS could also be implemented in other pub/sub systems. Other pub/sub systems may allow for simpler and more efficient implementations depending on the use cases.

3.6 Conclusion

This chapter has presented some ideas on how to solve the challenge. An architecture similar to the lambda architecture where each unique real-time stream has a dedicated publisher and a buffer-based transition is done in a client library has been chosen as the preferred one. Next chapter describes the proof-of-concept system architecture that incorporates the methods chosen here.

Chapter 4

System Architecture

This chapter describes a proof-of-concept system that implements the methods chosen in chapter 3. This system will be used for the testing in chapter 5.

The chapter starts with an overview of the system, its different components and how they interact. In the next sections, these main components are described in more detail.

4.1 Component Overview

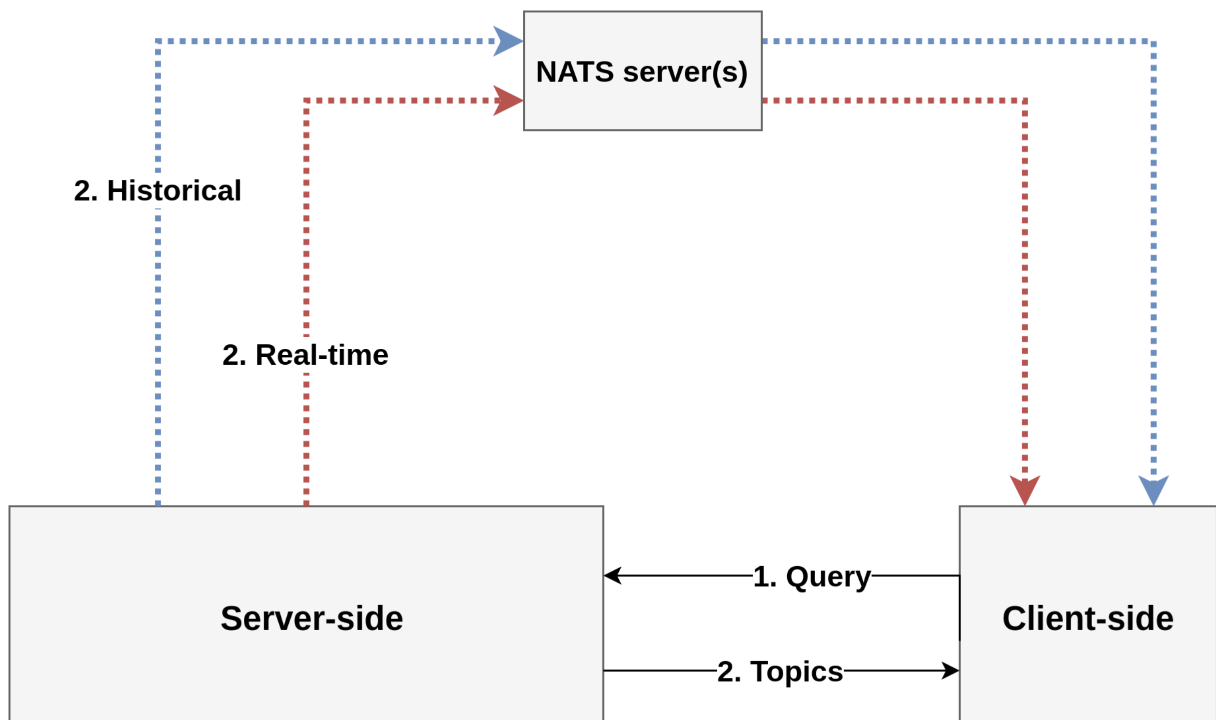


Figure 4.1: Information flow between the main components

The system consists of three main components: Server(s), the client(s), and a cluster of NATS servers. These can be further divided into smaller components. Figure 4.1 shows the main components of the system and the information flow between them.

The server-side represents a simplified version of Galore and consists of a gRPC server that receives and handles query requests from the clients and a simulator which starts up the required producers for each query.

The client-side consists of a client application that generates and sends the queries and a client library that handles the connection with NATS as well as the transition from historical to real-time topics.

4.1.1 Information Flow

Information flow for a query typically goes as follows: First, the client sends the query to the server. Next, the server figures out which historical and/or real-time topics the client should use and returns them to the client while simultaneously starting the publisher(s). The client subscribes to the assigned topics once it receives them and the messages are sent via the NATS cluster.

4.1.2 Queries and Messages

The queries used in this proof-of-concept system have been simplified to avoid the problem of having to figure out which queries result in the same real-time result. The queries contain an ID, a start timestamp, and an end timestamp.

The ID is analogous to the query string found in most query languages. In this system, it is just a random string of characters and does not carry any semantic meaning. The ID is used to represent which real-time stream it results in, so if two queries contain the same ID they will result in the same real-time stream. The timestamps are optional i.e., the lack of a start timestamp indicates that the query only uses real-time data, and a missing end timestamp indicates that the query continues onwards forever. This means that the queries can use either historical or real-time or both data streams.

The messages in this system represents a sensor measurement value at a specific point in time and consist of a timestamp and an integer. During the testing, the integer is set to be a counter value so that the client application can verify that the messages arrive in the correct order and that no messages are lost or sent twice.

The messages send as historical data also contain an End-of-File (EOF) tag. This is used to indicate whether or not the message is the last one in the database. When a message with an EOF tag is received, the client knows that there will not be any more historical messages until a new measurement is stored to the database.

Heartbeat Messages

Due to the pub/sub system acting as a layer of abstraction between the publisher and the subscribers, the server does not have a way to know whether or not there are any subscribers listening to its topics. Therefore each publisher regularly sends out a special heartbeat message. A heartbeat message does not contain any measurement data, but instead contains a reply-field to which the subscriber(s) have to reply. If no replies are given, the publisher can assume that its topic is inactive and stop publishing. This is implemented with the one-to-many request/reply messaging exchange in NATS.

4.2 Server-side

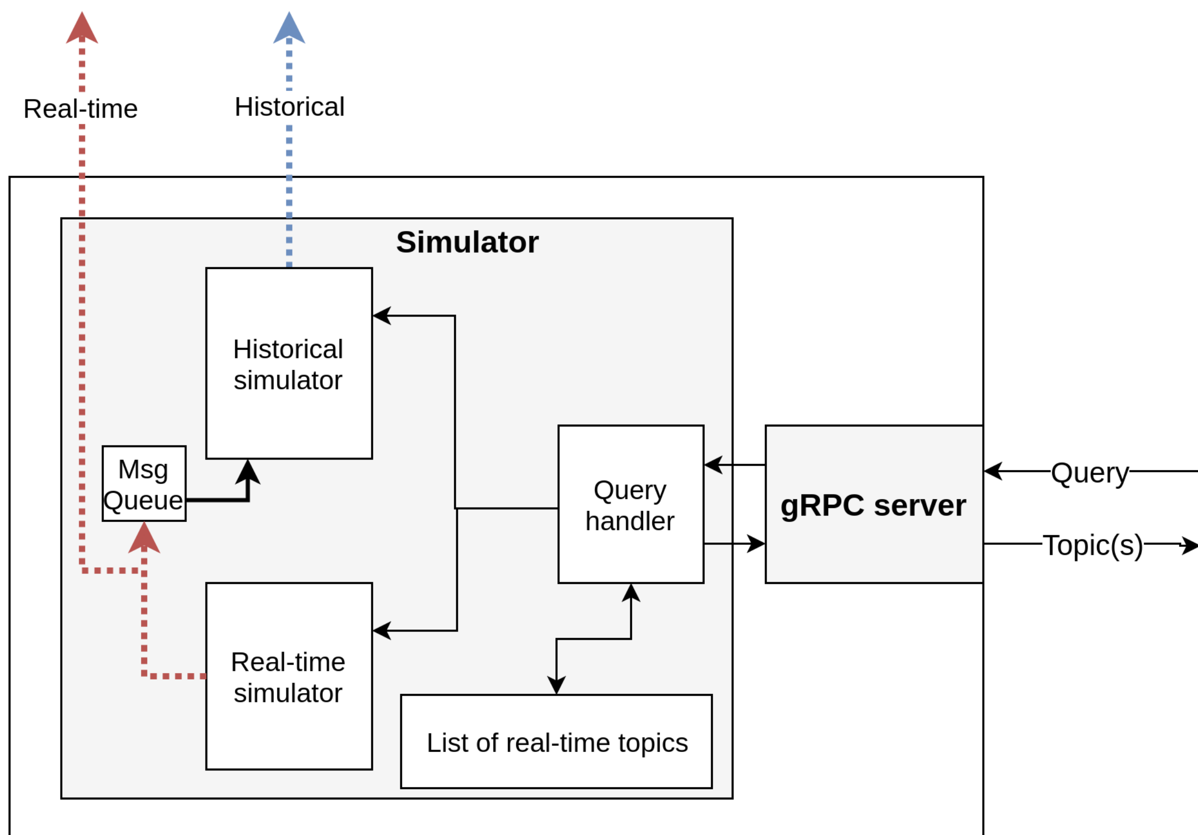


Figure 4.2: A more detailed illustration of the server-side.

The server-side represents a simplified version of Galore and consists of a gRPC server and a simulator. An illustration of the server-side and its components can be seen in figure 4.2.

4.2.1 The gRPC Server

The gRPC server handles the query requests and works as an interface between the clients and the simulator. The gRPC server listens for new queries from the clients and, when a query is received, it is forwarded to the simulator. The simulator returns the required topics which are then forwarded back to the client.

RPC and gRPC

An remote procedure call (RPC) system is a popular choice for communication between clients and servers in distributed systems [21]. RPC enables one machine to execute procedures on another machine as if it was a local procedure.

gRPC [22] is an open source RPC system. gRPC provides a simple way to specify the input parameters and the return types of the procedures and it handles all the details of the communication. This makes it a simple way to standardize the communication between the client-side and the server-side.

gRPC is not introduced in chapter 2 as it is only relevant for this implementation of the system, and not for the proposed architecture itself.

4.2.2 Simulator

The main responsibilities of the simulator are to assign topics to new queries and to simulate historical and real-time data streams. When a new query arrives, it checks the ID and timestamps to determine if it needs a historical and/or real-time simulator, and if a real-time publisher already exists for that ID. The real-time topic is determined by the query ID while the historical topic is a random string of characters, unique for each query. The simulator keeps track of which real-time topics are active so that it avoids starting duplicate publishers.

Real-time Simulator

The real-time simulator uses a pub/sub service to reach multiple clients with its messages without having to keep track of all the clients. A new message is published to the topic at an (approximately) constant interval.

A real-time query, theoretically, never ends. However, clients tend to disconnect for various reasons. Therefore, the publishers need a way to check if its topic still contains some active subscribers. To this, the publishers publish a special heartbeat message every 20 seconds to which the subscribers have to respond. If no subscribers respond within 3 seconds, the topic is assumed to be inactive and the simulator is stopped.

Historical Simulator

The historical simulator uses a one-to-one request-reply messaging pattern instead of pub/sub since the historical stream is not shared among multiple clients.

The historical publisher simulates the data coming from a database. The messages are sent as quickly as the client, NATS, and the simulator is able to process them. The timestamps of the messages are set so that they simulate the same interval as the real-time simulator uses. If this proof-of-concept system is to be further developed, the historical messages should preferably be sent in batches, but in this simulator, they are sent one by one.

If a query requires both historical and real-time data, the historical simulator also subscribes to the real-time topic so that it eventually publishes the same messages as the real-time simulator but with a small delay.

4.2.3 Making the Server-side Distributed

In order to test that the proposed solution can be built into a scalable environment, a simple gRPC server that can distribute the queries onto multiple simulators have been developed. Figure 4.3 shows a distributed setup of the system.

The only task of the master gRPC server is to forward the queries to the "slave" servers. When a new query is sent to the master gRPC server, its ID is hashed to find which slave server it is forwarded to. This ensures that all the queries that result in the same real-time stream are sent to the same slave server. The master then sends the query via a gRPC request to the slave server. The slave server returns the topic names which are then returned to the client.

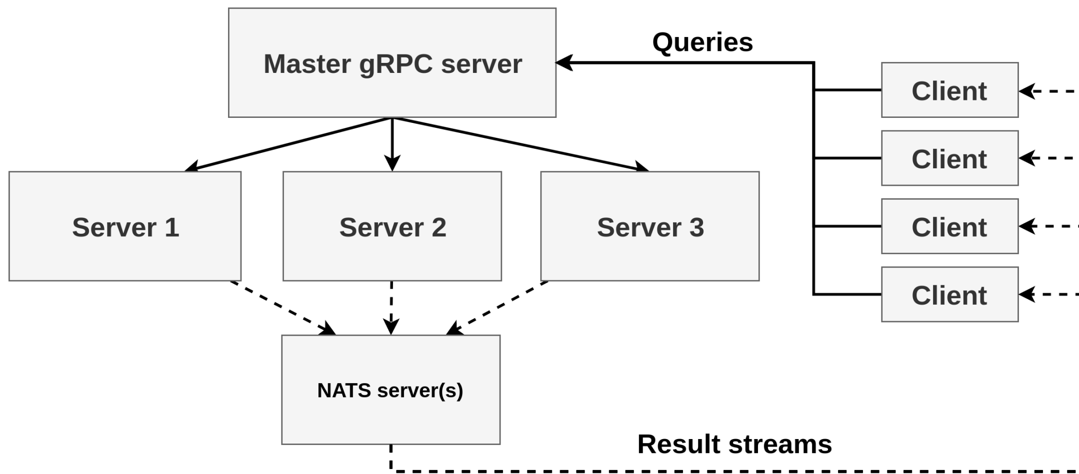


Figure 4.3: Using a master gRPC server to distribute the queries onto multiple servers.

A slave server is a normal instance of the "server-side" part of the system, consisting of a gRPC server and simulators. The slave server treats the requests coming from the master server the same way as a request coming from a client and does not need to know whether it is a standalone server or part of a distributed system.

4.3 Client-side

The client-side consists of a client application and a client library. An illustration of the client-side and its components can be seen in figure 4.4.

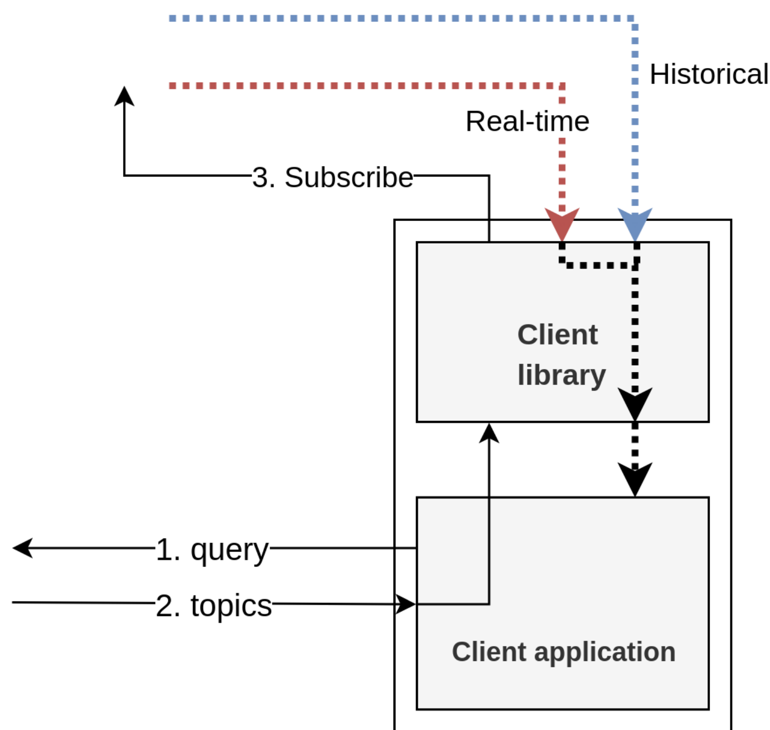


Figure 4.4: A more detailed illustration of the client-side

4.3.1 Client Application

The client application is the part of the system that generates and sends the query and uses the resulting stream. The query is sent to the gRPC server which returns the topics. A subscription-handler in the client library is called to handle the topics and to feed the result-stream back to the application.

Query Generator

A query generator has been developed to standardize the queries used for the testing in chapter 5 and to make sure that some of the IDs overlap. The number of overlapping query-IDs can easily be adjusted by command line arguments when multiple clients are started in the testing environment.

4.3.2 Client Library

The client library contains some useful tools for the client application. It is responsible for the transition from historical to real-time data and for the subscriptions and connection to the NATS cluster.

The Transition from Historical to Real-Time Stream

The transitioning from historical to real-time stream is done in three phases as shown in figures 4.5, 4.6, and 4.7.

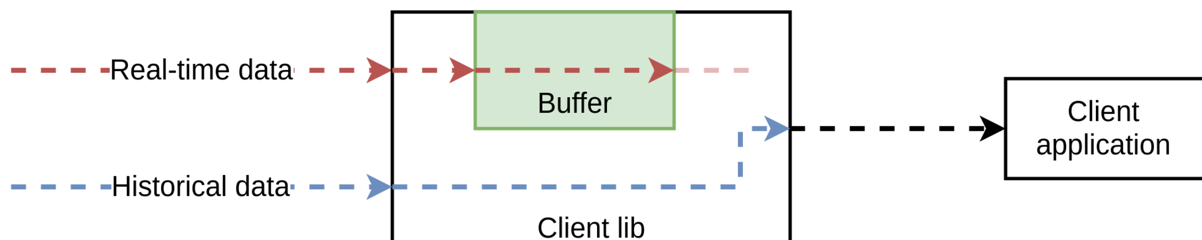


Figure 4.5: Transition phase 1: Forwarding historical stream.

In the first phase, shown in figure 4.5, historical data is forwarded directly to the client application while the real-time stream is sent to a buffer which stores the messages received in the most recent 30 seconds. More details about the data structure used as a buffer is found in appendix A.

This phase ends when an EOF message is received on the historical stream, indicating that the historical stream is simply waiting for new real-time messages before it can publish anything more. The client library can unsubscribe from the historical topic at this point.

The second phase, shown in figure 4.6, starts by iterating through the buffer to find the next message of the sequence. When the message is found, the rest of the buffer is output to the client application. It is assumed that the historical part of the query takes long enough for the buffer to fill in the gap between the two streams. The timestamp of the messages is used to find the correct order when comparing two messages. It is assumed that the timestamp is precise enough so that two messages from the same source do not have the same timestamp. The real-time stream keeps filling up the buffer while the transition is in this phase.

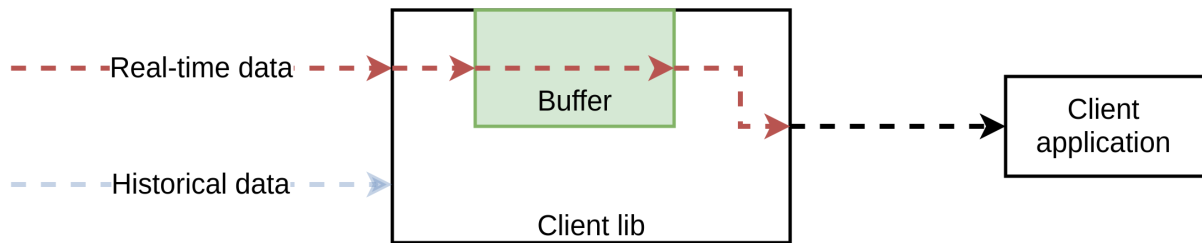


Figure 4.6: Transition phase 2: Forwarding from the buffer.

When the end of the buffer is reached, an attempt is done at transitioning from the buffer to the real-time stream. The last message read from the buffer is compared to the last message written to the buffer. If these messages are the same, it means that the next messages arriving on the real-time stream can be forwarded directly to the client. If they are not the same, the buffer is read again, starting from the next message and continuing until the end of the buffer before checking again. This is repeated until the messages match.

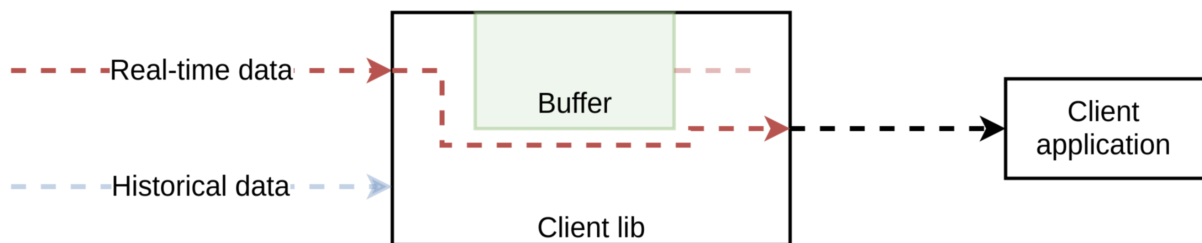


Figure 4.7: Transition phase 3: Forwarding from the real-time stream.

The third phase, shown in figure 4.7, starts with a successful transition from the buffer to a direct connection to the real-time stream. This is the last phase of the transition and the real-time stream is forwarded until the client application stops or disconnects.

Chapter 5

Testing and Results

To test the proposed solution, different scenarios and environments have been set up for the system.

As stated in section 1.1, the objective of this thesis is to scale a simplified model of Galore with regards to the number of clients while also handling the transition from historical to real-time data streams. Most of the components of the system have been tested through unit testing, including the method for transitioning from historical to real-time data streams. The tests of this chapter are therefore not focused on the functional aspect of each of the components, but rather how the system as a whole behaves in different scenarios.

How well the system handles an increased number of clients mainly depends on three aspects; what portion of the queries are able to share publishers, how well the server-side scales, and how well the pub/sub system works. NATS is not tested or benchmarked as the choice of pub/sub service is not in the scope of this thesis and the solution is designed to fit any pub/sub service.

Section 5.1 explains the environment used for the tests. In section 5.2, the system is tested with a varying portion of shared real-time streams. This is to test how effective the solution is for different usage patterns. Section 5.3 tests the scalability of the system to see how the system behaves when the server's load is distributed among multiple devices. Section 5.4 presents and analyzes the results of the tests. Finally, section 5.5 evaluates the planning, setup, and results of the tests.

5.1 Test Environment

To test the scalability of a distributed system, a test environment consisting of multiple nodes/servers are needed. Virtual machines (VMs) from Microsoft Azure [23] has been used for the testing as it makes it easy to assemble a system of any number of VMs while keeping the resources of each VM fixed so that similar results can be reproduced.

Some of the VMs are dedicated to act as clients. Instead of having each of the VMs running only a single client application, as they probably would in a real scenario, multiple client applications are run on the same VM for this test environment. The client application used in this proof-of-concept system is very light-weight and do not use the query result for anything other than checking that the order is correct. This means that a single VM can act as hundreds of clients at the same time, thereby simplifying the setup of the test environment significantly.

5.1.1 System Settings

Some of the settings are adjusted to make the system less stochastic so that the tests will be easier to reproduce.

- Even though the system is capable of handling queries that require either historical or real-time or both, this thesis is mainly focused on those that require both. Therefore, all queries start 100 seconds in the past and keep running forever. This increases the workload of the system for a short period following the arrival of a new query and it also ensures that the queries spend most of their lifetimes receiving real-time results.

- Each topic publishes a new value every second and the size of each message is constant. This means that each query receives at least 100 historical messages before transitioning to real-time where it gets a new message every second.

- Each client application sends 100 queries. When multiple client applications are started, the portion queries that results in the same real-time stream are adjusted according to the requirements of the test.

- Some extra computation has been added to the message generation procedure of the simulator. This is done to simulate some operation being done to get the result. It also simplifies the setup of the test environment as fewer client applications are needed to drain the resources of the simulator(s). The required computation is the same for all messages.

- Instead of sending a simulated measurement value to the clients, a counter value is sent when the system is used for testing. This means that the client application can check that no messages are lost during transmission or transition.

5.1.2 Performance Measures

Both of the tests use the CPU load as the metric to measure performance. The CPU load is the limiting resource with the current setup in both the tests and a good indication of how many queries and clients the server can handle. One of the benefits of using the CPU load compared to other metrics, e.g. latency and response times, is that it shows differences also when there are few clients.

The value is measured as an average percentage over a few minutes after all the queries have transitioned to real-time data.

5.2 Testing Different Usage Patterns

As the scalability improvement from this thesis comes from the ability to share publishers for equal queries, it is interesting to test how the system behaves when the portion of overlapping queries varies i.e., simulating differing usage patterns.

The average number of clients per publisher is used as the varying factor between the scenarios. The system is tested with an average of one, two, four, and ten clients per publisher. An average of one is the lowest possible and the worst case scenario. In this case, all the queries result in unique real-time results so that no publishers can be shared. An average of two means

that each publisher, on average, publishes its results to two clients. The server should therefore only need to use half the resources to support the same number of clients as in the previous scenario. Similarly, an average of four or ten means that the server, ideally, should use 25% or 10% of the resources it used in the first example, respectively.

5.2.1 Environment Setup

For this test, five VMs are used; one for NATS, one for the server, and three for clients. The VM running the server gets the least amount of resources while the other parts of the system get as many resources as they need. This is to make sure the server is the limiting factor of the test. The resources of the simulator VM is limited to 4GiB of RAM and two of the cores from an Intel Xeon CPU E5-2673 v3 with a clock speed of 2.40GHz. Figure 5.1 illustrates the system setup used in this test.

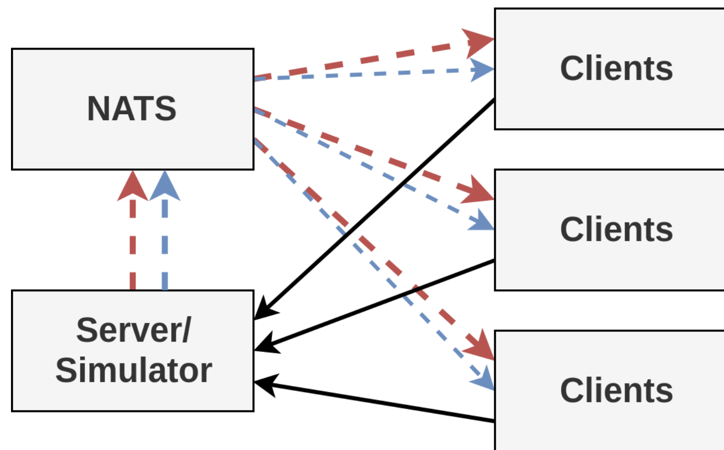


Figure 5.1: Environment setup for testing different usage patterns on a single server.

5.3 Testing the Scalability

Even if the system can save resources by sharing publishers, this benefit is of no use unless it also works when it is put into a scalable environment. A simple gRPC server has been developed to act as a master server that distributes the queries among multiple simulators. This section aims to test how many clients the system can handle when the queries are distributed among multiple servers.

5.3.1 Environment Setup

For this test, eight VMs are used; three are used for clients, one VM runs both the NATS server and the master gRPC server, and between one and four are used for slave servers. An illustration of the setup used in this test is shown in figure 5.2. Similarly to the previous test, the simulator(s) are given a restricted amount of resources while the other components are given however much they need. The VM(s) running the simulator(s) are limited to the same amount of resources as in the previous test.

The performance of the master gRPC server is not tested as it is only involved in assigning new queries to the simulators which is a much more lightweight task that the simulators.

This test measures how many unique real-time topics are required to reach a specific average CPU load among the servers. The system is tested for an average of 40%, 50%, and 60% CPU load when the work is distributed among one, two, three, and four servers.

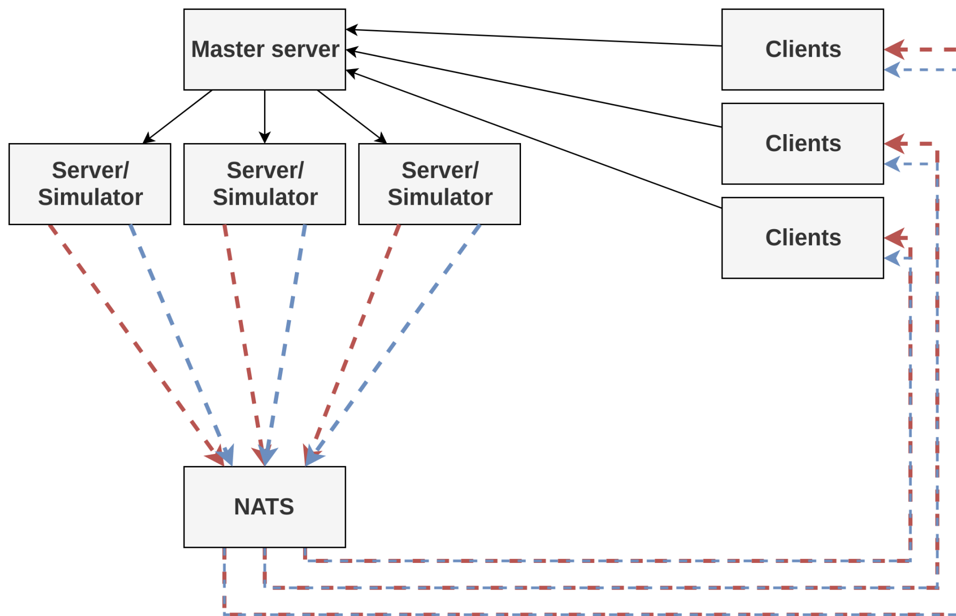


Figure 5.2: Environment setup for testing how the system scales. The master server and NATS were run on the same VM.

5.4 Results

This section presents and analyzes the results of the tests described in section 5.2 and 5.3.

5.4.1 Different Usage Patterns

This test is completed with four different average number of clients per publisher. Figure 5.3 shows the result of the test. A detailed table of all the measurements can be found in table B.1 in the appendix.

The X-axis shows the number of clients and the Y-axis shows the CPU load of the server. The blue, red, yellow, and green lines represent the four different usage patterns with the corresponding average number of clients per publisher of one, two, four, and ten. The system is only tested at a maximum of 1000 clients due to the VMs representing the clients not being able to handle more client applications (due to Azure enforcing some operating system limits on the maximum number of open file descriptors).

The results in figure 5.3 show a clear increase in the supported number of clients when more queries overlap, but the differences between the four scenarios vary a bit compared to expected. Especially the yellow line is steeper than expected up till the first 350 clients.

E.g., when studying the 30% line, the blue line crosses it at approximately 70 clients. The corresponding expectation for the other lines is then 140, 280, and 700. The red line reaches 30% at 150 clients, the yellow line at 230 clients, and the green line at 600.

For the 50% line, the blue line crosses it at approximately 150 clients. The ideal result would then be for the other lines to cross the same line at 300, 600, and 1500 clients. The red line crosses the line at 270 clients and the yellow line crosses it at 590 clients. The scenario with 10 average clients per publisher is not tested at high enough client count to verify this.

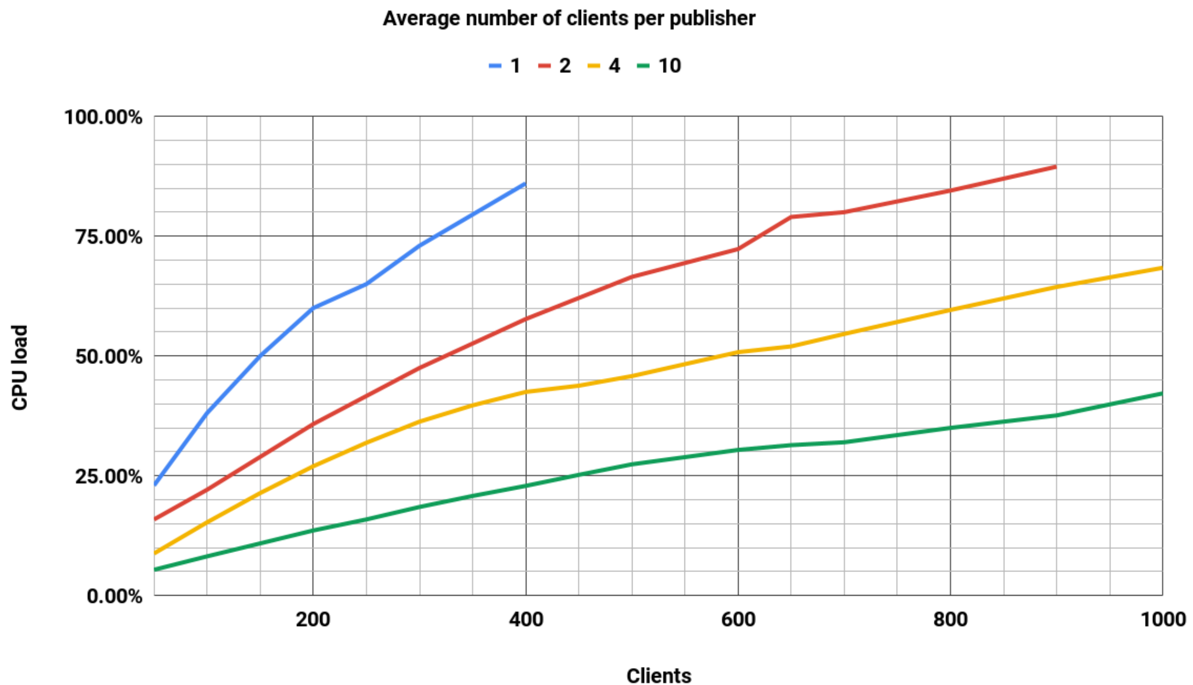


Figure 5.3: Result of testing different usage patterns. The blue, red, yellow, and green line shows how the CPU load increases as the number of clients increase when the average number of clients per publisher is one, two, four, and ten.

5.4.2 Scalability

The test has been done by sharing the publishers between one, two, three, and four servers. Figure 5.4 shows the result of the test. The X-axis shows the number of servers used and the Y-axis shows the number of clients required to reach a specific average CPU load. The red, blue, and green dots show the results for 40%, 50%, and 60%, respectively. The corresponding red, blue, and green lines show the expected result for a perfect linear scale based on the results for a single server.

A detailed table of all the measurements from this test can be found in table B.3 the appendix.

As the figure shows, the results are very close to the linear line for all three CPU load percentages. This means that, for example, doubling the number of servers results in the system supporting approximately twice as many publishers.

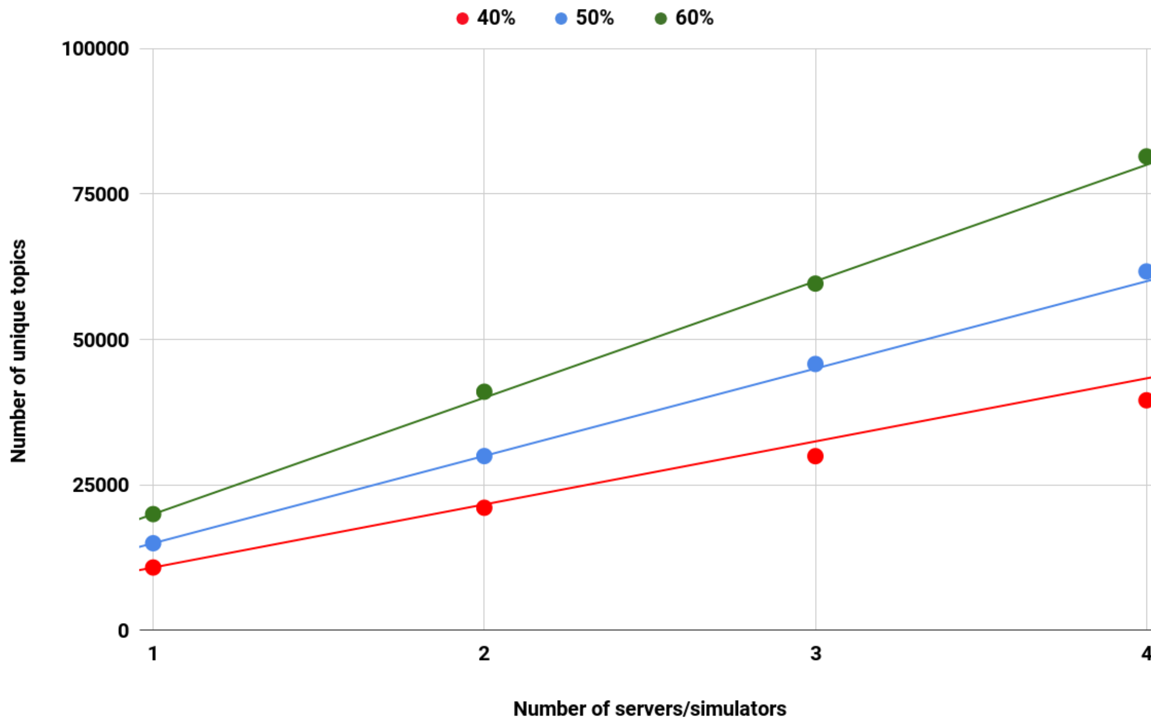


Figure 5.4: Results from testing the scalability. The plot shows how many unique topics it takes to reach an average CPU load of 40%, 50%, and 60% for the servers, illustrated by the green, blue, and red dots. Linear lines are drawn for comparison.

5.5 Evaluation

This section evaluates the testing and the results from the previous sections and puts it in the context of the requirements introduced in chapter 1.

The Test Environment

The tests presented in this chapter is done with a proof-of-concept system in an artificial environment, meaning that not all the aspects of the test setup are realistic.

For example, the communication cost of the test environment is expected to be much lower than in a real scenario as all parts of the system (including the clients) are in close proximity to each other. Also, many of the aspects of the system are heavily simplified compared to a real implementation.

The results are, however, still interesting, but the focus of these tests should be on the relative changes and differences in measurements rather than the specific client counts or CPU load percentages.

Meeting the Requirements

The goal of the testing is to check if the proposed solution meets the requirements from chapter 1. The overall goal of the thesis is to scale a simplified model of Galore for an increased number of clients by the use of a pub/sub service. The solution should be such that it only uses one producer per group of queries that results in the same real-time stream and that the transition is

abstracted away from the user application.

The test of usage patterns in section 5.2 is meant to show that when multiple queries result in the same real-time stream and thereby uses a shared producer, the system is able to handle more clients. The results show that the number of supported clients increases as more of the queries result in the same real-time stream. The performance differences between the scenarios do have some variations between what is measured and what is expected.

Another way to look at the extent of the difference between the expected and the measured CPU load is to look at the CPU load versus the number of publishers. Figure 5.5 shows the result of the four scenarios but looks at the number of publishers instead of the number of clients. The expected result is that all four scenarios should require the same amount of resources for the same amount of producers. As the figure shows, there are some cases where the CPU load varies with up to 10%, but for most of the measurements, the variation is less than 5%. This variation could be caused by e.g., inaccurate measurements, other background tasks being run on the VMs, or by some errors in the implementation.

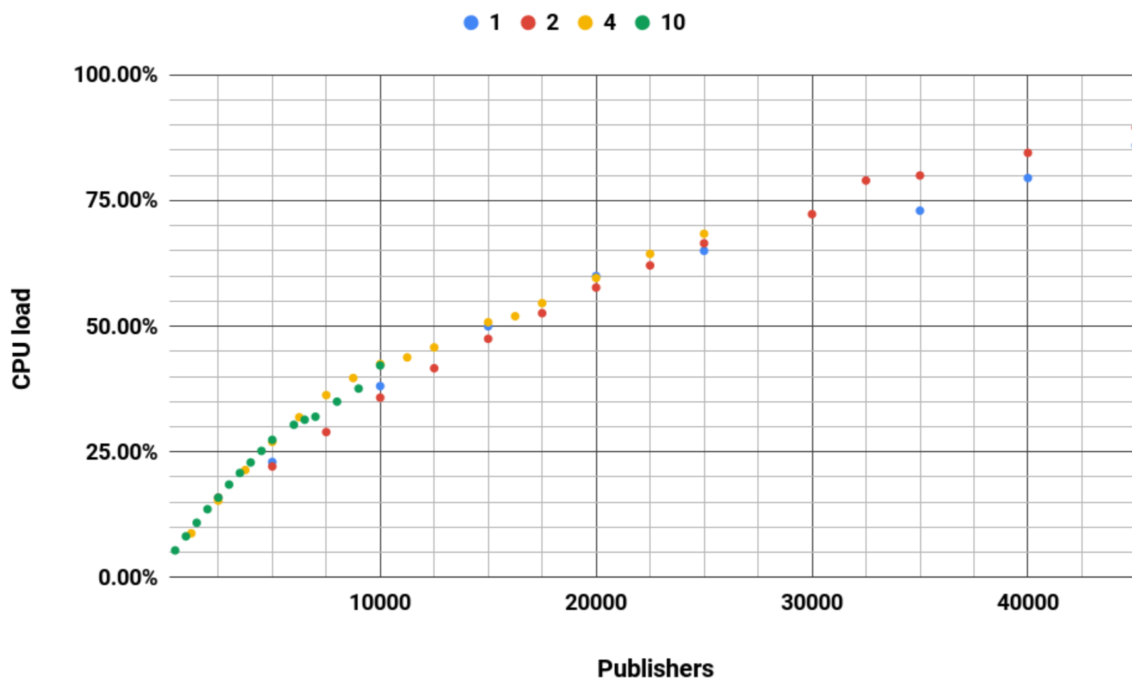


Figure 5.5: Results from testing different usage patterns, but normalized to the number of unique real-time result streams. A detailed table of the results can be found in table B.2 in the appendix.

Due to the VMs running the client applications being restricted to a total of approximately 1000 clients (or 100 000 queries), the test is missing some data on the highest CPU percentages for the scenarios with four or ten clients per publisher. This means that some of the comparisons can only be done on two or three of the scenarios, making them less accurate.

The requirement of only having one publisher per group of queries that result in the same real-time stream is intrinsic to the system by design. The simulator keeps a list of the topics that it is publishing to so that it doesn't spin up a new publisher where one already exists. Any new

query that already has a real-time result simulation running will simply connect to that one.

For the testing, the server regularly outputs the number of queries and the number of publishers, making it easy to spot if the numbers are wrong. The client applications would also notice if messages came from multiple publishers due to the messaging interval being wrong. The system kept the correct number of publishers throughout the testing thus, this seems to work as expected.

The scalability test in section 5.3 is meant to show that the proposed solution can easily be put in a scalable system and that it scales well. The results show that the system scales more or less linearly and proves that the system can be made scalable fairly easy, at least for a few servers.

The system is only tested with a maximum of four servers and thus, the tests do not give any insight into how well or for how long it will keep on scaling. Testing a distributed system at a large scale requires a large and often expensive environment, something that was not available in this project. If more servers are needed, eventually the bottleneck is likely to be the master gRPC server. The current master gRPC server is very simple and only meant to show that the proposed solution fits into a distributed system. Making a more advanced master server that handles large scaling is outside the scope of this thesis.

The scalability is only tested for the scenario where all the queries require their own publisher. This is because the test with different usage patterns showed that the number of clients doesn't affect the CPU load unless their queries require their own publishers. Figure 5.5 shows that the CPU load is more or less the same for all four scenarios when only considering the number of publishers.

The simulators are exactly the same for the distributed version as in the standalone version, the only difference is that the queries all come from the master gRPC server which distributes the incoming queries among the slave simulators. It is therefore assumed that similar results could be achieved for the various usage patterns also when the system consists of multiple servers.

The transition from historical to real-time data does not have its own test in this chapter. The functionality is however tested in a unit test. In addition to this, it is also being constantly monitored for faults throughout the other tests in this chapter. This is the reason why all the queries used for the tests in this chapter started 100 seconds in the past. No faults were found during this testing.

The other requirement for the transition is that it is abstracted away from the user application. This is already accomplished by the design of the system as it is located in a client library. The user application only needs to provide the topics returned by the gRPC server and the client library handles both the connection with NATS and the transitioning.

Use of CPU Load as the Metric

The CPU load of a machine gives a good indication of how much computational resources the running processes require. This is mainly suited as a metric for machines that only perform a single task. The VMs running the servers in these tests uses most of their computational resources on the server processes and thus, it is a good fit. A small portion of resources is used on other background tasks orchestrated by the operating system, but this portion is very small compared to the resource requirements of the server.

Conclusion

Throughout this chapter, the proof-of-concept system has been tested to see if it meets the requirements specified in chapter 1.

Two tests have been conducted; a test of how efficient the solution is in different usage patterns and a test of how the system scales. The results of both these tests show that the system behaves very similar to what was expected. The server is able to support a lot more clients when the publishers can publish their results to multiple clients and the solution can successfully be implemented in a scalable system.

The requirement of having only one publisher for all queries that result in the same real-time stream is achieved by the systems architecture and the requirement to abstract the transitioning away from the client application is handled by a client library.

All the requirements are therefore met.

Chapter 6

Conclusion and Future Work

This chapter concludes the work done in this thesis and presents some ideas for future work on the subject.

6.1 Conclusion

The goal of this thesis was to find a method that allows queries to share a publisher when they result in the same real-time stream by the use of a pub/sub system. At the same time, the task of transitioning from historical to real-time streams should not be left for the client application to solve. The point of this was to improve the scalability of the IIoT system Galore for an increased number of clients.

Different approaches to the challenge have been discussed and an architecture has been proposed based on the requirements. This architecture borrows ideas and techniques from Multi-Query Optimization, the lambda architecture, and pub/sub systems to meet the requirements.

Finally, a proof-of-concept system has been implemented to test the architecture. The first test shows that the system can handle a lot more clients when the publishers can share their results with multiple clients at once. The second test shows that the architecture can easily be applied to a distributed environment and make it scale horizontally.

The testing shows that the requirement of sharing publishers for queries that result in the same real-time stream is met. At the same time, the requirement for the transition from historical to real-time data being abstracted away from the client application is achieved by delegating the task to a client library. Based on the result from the testing it is likely that the proposed architecture will indeed improve the scalability of Galore for an increased number of clients.

6.2 Future Work

The proposed architecture and the testing done in this thesis is just one of the challenges associated with scaling the client-handling capabilities of Galore.

The biggest remaining challenge is probably to find an efficient method of analyzing the incoming queries to determine which ones result in the same real-time result streams. The methods proposed in this thesis does not add value to a system unless queries that share result streams can be found. This task is highly dependant on the query language in use and how the

stream processing is implemented. In Galore's case, this is a proprietary language called TQL.

Performing a thorough comparison of different pub/sub systems could improve the performance and simplify the development if the proposed architecture is to be implemented in a bigger system. The different pub/sub systems have vastly different features and use cases.

If this architecture is to be used in a scalable environment, a more sophisticated master gRPC server should be developed. The current master is very simplistic and not suited for a production environment.

If the queries that require both historical and real-time data spend a lot of time consuming historical data, it could be beneficial to develop a method to delay the subscription to the real-time topic. The real-time messages must cover the delay between the two streams when the transition happens, but any real-time messages before this is a waste of bandwidth resources. This could, for example, be done by predicting what time stamp the last historical message will have based on the transfer rate and time stamps of the historical messages.

Bibliography

[1] *KONGSBERG DIGITAL*. [Online], downloaded 2019-05-06. URL: https://www.kongsberg.com/digital.

[2] *Galore/TQL Syntax*. [Online], downloaded 2019-05-06. URL: https://github.com/kognifai/Galore/blob/master/Galore-Documentation/TQL%5C%20Syntax.md.

[3] Prasan Roy and S. Sudarshan. "Multi-Query Optimization". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 1849–1852. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_239. URL: https://doi.org/10.1007/978-0-387-39940-9_239.

[4] Timos K. Sellis. "Multiple-query Optimization". In: *ACM Trans. Database Syst.* 13.1 (Mar. 1988), pp. 23–52. ISSN: 0362-5915. DOI: 10.1145/42201.42203. URL: http://doi.acm.org/10.1145/42201.42203.

[5] Jooseok Park and Arie Segev. "Using Common Subexpressions to Optimize Multiple Queries". In: *Proceedings of the Fourth International Conference on Data Engineering*. Washington, DC, USA: IEEE Computer Society, 1988, pp. 311–319. ISBN: 0-8186-0827-7. URL: http://dl.acm.org/citation.cfm?id=645473.653403.

[6] Wenfei Fan, Jeffrey Xu Yu, Hongjun Lu, et al. "Query Translation from XPATH to SQL in the Presence of Recursive DTDs". In: *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB '05. Trondheim, Norway: VLDB Endowment, 2005, pp. 337–348. ISBN: 1-59593-154-6. URL: http://dl.acm.org/citation.cfm?id=1083592.1083634.

[7] Hoshi Mistry, Prasan Roy, S. Sudarshan, et al. "Materialized View Selection and Maintenance Using Multi-query Optimization". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 307–318. ISBN: 1-58113-332-4. DOI: 10.1145/375663.375703. URL: http://doi.acm.org/10.1145/375663.375703.

[8] Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, et al. "Efficient Exploitation of Similar Subexpressions for Query Processing". In: *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*. SIGMOD '07. Beijing, China: ACM, 2007, pp. 533–544. ISBN: 978-1-59593-686-8. DOI: 10.1145/1247480.1247540. URL: http://doi.acm.org/10.1145/1247480.1247540.

[9] Sailesh Krishnamurthy, Chung Wu, and Michael Franklin. “On-the-fly Sharing for Streamed Aggregation”. In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’06. Chicago, IL, USA: ACM, 2006, pp. 623–634. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142543. URL: http://doi.acm.org/10.1145/1142473.1142543.

[10] Matthew Denny and Michael J. Franklin. “Predicate Result Range Caching for Continuous Queries”. In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’05. Baltimore, Maryland: ACM, 2005, pp. 646–657. ISBN: 1-59593-060-4. DOI: 10.1145/1066157.1066231. URL: http://doi.acm.org/10.1145/1066157.1066231.

[11] N. Marz and J Warren. *Big Data: Principles and Best Practices of Scalable Realtime Data Systems*. eng. Stamford: Manning Publications Co, 2015. ISBN: 1617290343.

[12] Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. “Survey of Real-time Processing Systems for Big Data”. In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. IDEAS ’14. Porto, Portugal: ACM, 2014, pp. 356–361. ISBN: 978-1-4503-2627-8. DOI: 10.1145/2628194.2628251. URL: http://doi.acm.org/10.1145/2628194.2628251.

[13] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, et al. “The Many Faces of Publish/Subscribe”. In: *ACM Comput. Surv.* 35.2 (June 2003), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: http://doi.acm.org/10.1145/857076.857078.

[14] David S. Rosenblum and Alexander L. Wolf. “A Design Framework for Internet-scale Event Observation and Notification”. In: *SIGSOFT Softw. Eng. Notes* 22.6 (Nov. 1997), pp. 344–360. ISSN: 0163-5948. DOI: 10.1145/267896.267920. URL: http://doi.acm.org/10.1145/267896.267920.

[15] M. Altherr, M. Erzberger, and S. Maffeis. “iBus—A software bus middleware for the Java platform”. In: *Proceedings of the International Workshop on Reliable Middleware Systems of the 13th IEEE Symposium On Reliable Distributed Systems (SRDS)*. 1999, pp. 43–53.

[16] Ian G. Craggs. *Managing topical overlap during publication and subscription*. US Patent: US8849754B2. Assignee: International Business Machines Corporation. Oct. 2005. URL: https://patents.google.com/patent/US8849754B2/en.

[17] *Subject-based Messaging*. [Online], downloaded 2019-05-30. URL: https://nats.io/documentation/writing_applications/subjects/.

[18] *Apache Kafka. A distributed streaming platform*. [Online], downloaded 2019-05-05. URL: https://kafka.apache.org/.

[19] *RabbitMQ*. [Online], downloaded 2019-05-05. URL: https://www.rabbitmq.com/.

[20] *NATS - Open Source Messaging System*. [Online], downloaded 2019-05-05. URL: https://nats.io/.

[21] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018. URL: http://www.ostep.org.

[22] *A high performance, open-source universal RPC framework*. [Online], downloaded 2019-06-3. URL: https://grpc.io/.

[23] *Microsoft Azure*. [Online], downloaded 2019-05-01. URL: https://azure.microsoft.com.

Appendix A

Sliding Buffer Data Structure

For a client to store the messages from the last X seconds of the real-time stream, a buffer data structure has been developed.

The buffer consists of a list of buckets. Each of these buckets represents a time interval of a fixed length, starting from the timestamp of its oldest message, and contains a list of all the messages received within that time interval. Figure A.1 shows the main components of the structure.

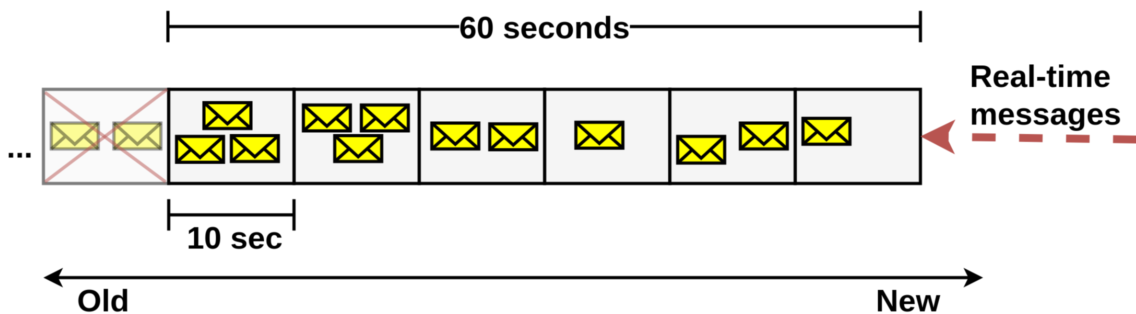


Figure A.1: A sliding buffer consisting of 6 buckets of 10 seconds each. The leftmost bucket has just been removed.

In the case of the figure, there are 6 buckets, each representing 10 seconds. This means that the buffer effectively stores between 50 and 60 seconds of the stream depending on how long its been since the last shift.

For a new message, the algorithm works as follows:

1. Find the newest bucket.
2. If the bucket's starting timestamp is less than 10 seconds older than the timestamp of the new message:
    2.1. Add the message to the bucket.
3. Else:
    3.1. Make a new bucket.
    3.2. Add the message to the new bucket.
    3.3. Add the new bucket and remove the oldest bucket from the list of buckets.

The use of buckets instead of just having a long list of messages means that the shift operation happens a lot less frequently. It also allows for more efficient searches when it is time for the transition. Instead of searching through a long list of messages to find out where to start reading, one can first search through a smaller list of buckets and then search through the messages of that bucket to find out where to start.

A buffer based on a fixed time interval instead of a fixed number of messages means that the time window can be adjusted to fit the delay between the historical and the real-time stream.

If the processes being applied to the historical streams and the real-time streams are very different, the delay between them might vary for each query. For this case, it might be better to use a buffer based on a fixed number of messages instead. This is simply a matter of shifting the buckets when they reach a specified count instead of a specified duration.

Appendix B

Raw Test Results

Different Usage Patterns

This table contains the raw results of the test for different usage patterns described in section 5.2.

Clients	1 (all unique)	2	4	10
50	23.0%	15.9%	8.8%	5.4%
100	38.1%	22.1%	15.3%	8.2%
150	50.0%	28.9%	21.4%	10.9%
200	60.0%	35.8%	27.0%	13.6%
250	65.0%	41.6%	31.9%	15.9%
300	73.0%	47.5%	36.3%	18.5%
350	79.5%	52.6%	39.7%	20.8%
400	86.0%	57.7%	42.5%	22.9%
450		62.1%	43.8%	25.2%
500		66.5%	45.8%	27.4%
600		72.3%	50.8%	30.4%
650		79.0%	52.0%	31.4%
700		80.0%	54.6%	32.0%
800		84.5%	59.6%	35.0%
900		89.5%	64.4%	37.6%
1000			68.4%	42.2%

Table B.1: The test results for different usage patterns

CPU Load vs Number of Publishers

This table contains the raw results used in figure 5.5. These are the same measurements as in figure B.1, but the rows now show the CPU load vs the number of publishers instead of the number of clients.

	1	2	4	10
500				5.40%
1000				8.20%
1250			8.80%	
1500				10.90%
2000				13.60%
2500		15.90%	15.30%	15.90%
3000				18.50%
3500				20.80%
3750			21.40%	
4000				22.90%
4500				25.20%
5000	23.00%	22.10%	27.00%	27.40%
6000				30.40%
6250			31.90%	
6500				31.40%
7000				32.00%
7500		28.95%	36.30%	
8000				35.00%
8750			39.70%	
9000				37.60%
10000	38.10%	35.80%	42.50%	42.20%
11250			43.80%	
12500		41.65%	45.80%	
15000	50.00%	47.50%	50.80%	
16250			52.00%	
17500		52.60%	54.60%	
20000	60.00%	57.70%	59.60%	
22500		62.10%	64.40%	
25000	65.00%	66.50%	68.40%	
30000		72.30%		
32500		79.00%		
35000	73.00%	80.00%		
40000	79.50%	84.50%		
45000	86.00%	89.50%		

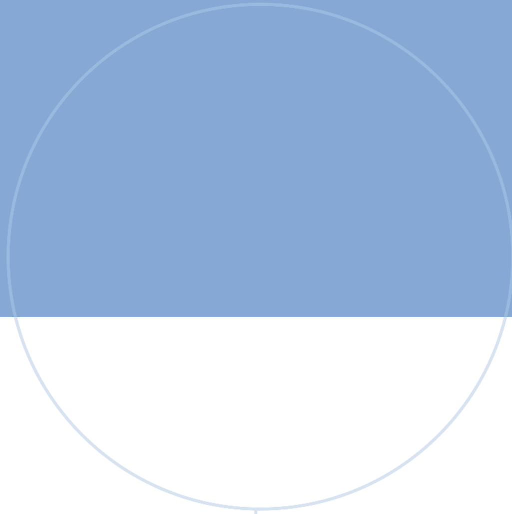
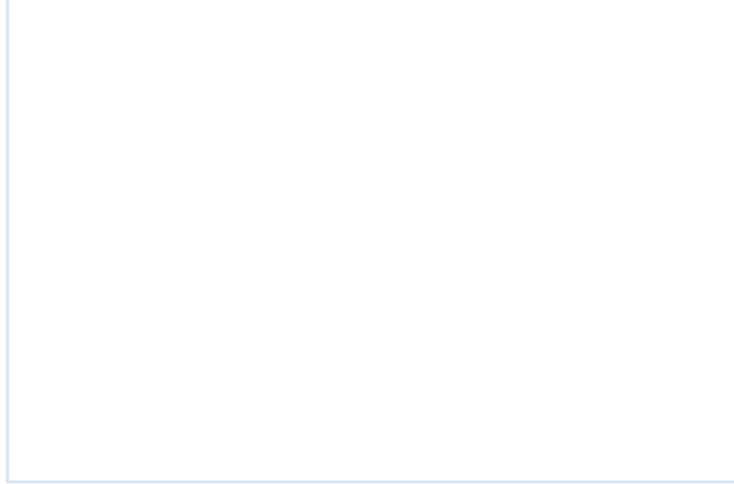
Table B.2: CPU usage for the different usage patterns vs. the number of publishers.

Results from Testing the Scalability

This table contains the measurements used in figure 5.4.

Number of servers:	**40%**	**50%**	**60%**
1	10830	15000	20000
2	21094	29970	41024
3	29955	45788	59602
4	39553	61685	81446

Table B.3: Number of clients required to reach 40%, 50%, and 60% CPU load when the queries are distributed among multiple servers.

NTNU
Norwegian University of
Science and Technology

KONGSBERG