Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science - Software Engineering 30.0 credits

A COMPARISON OF DATA INGESTION PLATFORMS IN REAL-TIME STREAM PROCESSING PIPELINES

Sebastian Tallberg
stg19001@student.mdh.se

Examiner: Marjan Sirjani
Mälardalen University, Västerås, Sweden

Supervisors: Wasif Afzal
Mälardalen University, Västerås, Sweden

Company supervisor: Amel Muftic,
Addiva AB, Västerås, Sweden

June 7, 2020

Abstract

In recent years there has been an increasing demand for real-time streaming applications that handle large volumes of data with low latency. Examples of such applications include real-time monitoring and analytics, electronic trading, advertising, fraud detection, and more. In a streaming pipeline the first step is ingesting the incoming data events, after which they can be sent off for processing. Choosing the correct tool that satisfies application requirements is an important technical decision that must be made. This thesis focuses entirely on the data ingestion part by evaluating three different platforms: Apache Kafka, Apache Pulsar and Redis Streams. The platforms are compared both on characteristics and performance. Architectural and design differences reveal that Kafka and Pulsar are more suited for use cases involving long-term persistent storage of events, whereas Redis is a potential solution when only short-term persistence is required. They all provide means for scalability and fault tolerance, ensuring high availability and reliable service. Two metrics, throughput and latency, were used in evaluating performance in a single node cluster. Kafka proves to be the most consistent in throughput but performs the worst in latency. Pulsar manages high throughput with low message sizes but struggles with larger message sizes. Pulsar performs the best in overall average latency across all message sizes tested, followed by Redis. The tests also show Redis being the most inconsistent in terms of throughput potential between different message sizes.

Table of Contents

List of Figures

List of Tables

1 Introduction

Real-time data processing is a problem that has been worked on since the 1990s [11]. As the amount of data being produced has increased, coupled with increasingly complex software solutions being developed, there is a need for platforms that address these needs. Streaming applications such as fraud detection, network monitoring and electronic trading rely on real-time data processing to ensure that the service provided is deemed correct and reliable.

The vast majority of modern applications use some sort of database management system for handling data. As data is gathered or produced by the application, it is stored and indexed such that it can be queried later on by the application. However, for applications that have stricter requirements on real-time data processing this is not a suitable approach. This is where stream processing comes into play.

Stream processing is about processing the data directly as it is received. Real-time stream processing applications often have certain key requirements that must be fulfilled. Having low latency between the input and the processed data output is a key characteristic in enabling real-time applications [12]. A more traditional batch-processing approach requires gathering data in so called batches, where the processing can begin only once the final piece of data of each batch has arrived. For real-time use cases, the delay this causes is unacceptable, as the latency in these real-time streaming applications should preferably be within milliseconds. For example, in electronic trading a delay of even 1 second is deemed intolerable. These applications often also require high throughput, i.e. allow for processing of large volumes of data. Additional key features of stream processing platforms include data safety and availability, handling data that is out of order or missing, storage system support, and more [13].

Traditionally, custom solutions were being developed by the companies themselves to address the requirements of real-time processing. This mostly resulted in inflexible solutions with a high development and maintenance cost [13]. Today, however, there exists several stream processing platforms and frameworks that address these requirements to various degrees. These technologies keep continuously evolving by introducing new features and improving performance.

In a real-time stream processing pipeline, two of the major components are data ingestion and processing. This thesis will focus on evaluating and comparing three open-source technologies that address the data ingestion part of the pipeline; Apache Kafka [14], Apache Pulsar [15] and Redis Streams [16]. Understanding which platform to choose for data ingestion based on the characteristics and performance of the system is important when developing real-time streaming applications. The thesis work is a collaboration with Addiva AB [17] in an attempt to assess these factors in the context of a real-world software project.

1.1 Problem Formulation

The objective of the thesis is to evaluate the selected platforms that solve the data ingestion part of the real-time data pipeline. The evaluation of the different platforms is done based on certain *characteristics* and *performance* metrics that the company is interested in. The *characteristics* used for evaluation are data persistence and retention, fault tolerance, language and container support. A high-level architecture overview will also be provided for each of the selected platforms. In terms of the *performance* evaluation, throughput and end-to-end latency are the two metrics that are of interest. For measuring the maximum throughput, benchmarking tools for respective platforms will be used. Measuring the end-to-end latency will be done by creating a test program for each of the platforms, where the timestamp of the message when it was sent is compared to the timestamp of when it is received. The parameters affecting the performance, i.e. message size and messages sent per time unit, are determined by the company based on their real-world needs. Based on the objective of the thesis, the following research questions have been derived:

RQ 1: What are the differences in the identified characteristics of the chosen platforms?

RQ 2: What are the differences in performance metrics of the chosen platforms?

2 Background

The following section aims to present the basic terminology and concepts related to the data ingestion part of a real-time stream processing pipeline. A high-level architectural overview is also given for Redis, Kafka and Pulsar, which are the platforms selected for comparison.

2.1 Event stream

The fundamental concept to grasp in stream processing is the stream itself. Many different terminologies have been used depending on the domain; for example event stream, data stream and event sourcing [11]. In the context of this work, the stream will be referred to as an event stream. The event stream can be described as an append-only log, i.e. the events that are captured are immutable and in fixed order. For example, the popular stream processing platform Apache Kafka uses this approach, as seen in *figure 1.* In a real-time streaming application the event stream is likely to be a sequence of unbounded events, i.e. the incoming events are infinite with no expected end. An event could for example be a sensor reading, credit card transaction, web search or weather station observation [12].

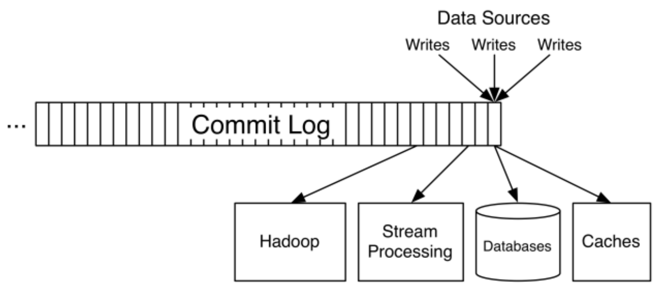

Figure 1: Apache Kafka commit log of events [1]

2.2 Publish/subscribe

Publish/subscribe is one of the most well-known messaging patterns used to communicate data between a sender (publisher) and a receiver (subscriber) [18]. Instead of sending the messages directly between each other, a broker is most often used to facilitate the communication. The publishers send messages to so-called topics in the broker, which are used to separate different types of data that are being communicated. The broker is responsible for correctly routing each message to the subscribers of a topic. Each topic can have several subscribers, and the incoming messages will be delivered to all of them. Figure 2 visualizes the publish/subscribe pattern.

While the terminology used in systems such as Apache Kafka, Apache Pulsar and Redis Streams are slightly different, they are all based on this publish/subscribe type of communication. They also offer more advanced platform specific features that extends the foundation of publish/subscribe messaging.

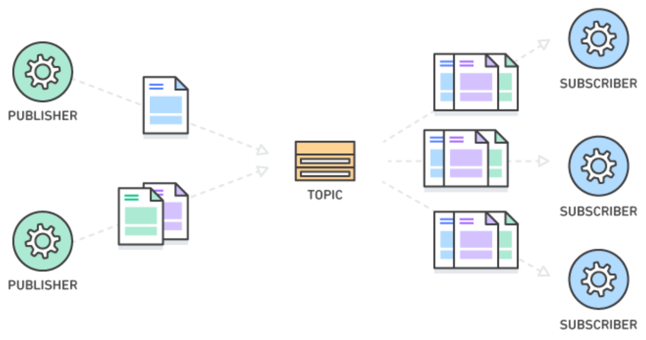

Figure 2: Publish/subscribe pattern [2]

2.3 Performance metrics

In comparison to more traditional software, streaming applications have certain requirements that must be fulfilled. Since the event stream is unbounded, the application must be able to process the events as quickly as possible, as well as be able to handle a high rate of incoming events. Therefore, the two main metrics that are used for evaluating performance of such systems are latency and throughput [12].

Latency is a measurement of the delay between when the event has been received and when it has successfully been successfully processed. Latency in streaming systems is measured in milliseconds. Depending on the type of application one might be interested in either the average latency or maximum latency. Low latency is the key requirement in many streaming applications as it enables the processing of data in real-time. [12]

Throughput refers to the rate of events that can be processed during a certain time period. In contrast to latency, the maximum possible throughput should be as high as possible. If the incoming events are arriving at a faster rate than the system can process them, the events will be buffered and thus negatively affecting the latency [12]. Thus, it might often be a trade-off between throughput or latency based on how the system is configured. Stream processing systems are usually categorized as either scale-out or scale-up. Scale-out systems take advantage of distributed processing over a large amount of nodes, while scale-up systems try to take advantage of a single high-performance machine [19].

2.4 Redis Streams

Redis is an in-memory key-value store that has had widespread use as a database, cache and message broker [20]. Previously, it has been possible to do simple stream processing using Redis' implementation of the publish/subscribe messaging paradigm, using Redis itself as the broker. The publisher could for example be the data source, which continuously sends streams of data to a specific channel. One or more subscribers can then subscribe to this channel and consume the data as it arrives. Figure 3 shows an example twitter data processing pipeline using Redis' publish/subscribe system.

A problem with this approach is data loss, as there is no data persistence. Once the data in the channel has been sent out and consumed by the subscriber(s), that data is lost. The implication of this is that historical data cannot be revisited at a later time. Additionally, data loss will also occur if the subscriber loses connection with the broker, as Redis publish/subscribe does not guarantee message delivery to its subscribers. These are often key requirements that are wanted in a streaming application.

Redis version 5.0 came with the introduction of Redis Streams [16], a new data structure that aims to alleviate the shortcomings of the previously mentioned solution. The stream data structure resembles an append only log, where new entries are added to the end of the stream. A Redis server can contain multiple stream structures, which can be identified by giving them a unique name. Producers are the components responsible for writing data to the stream. For example, a

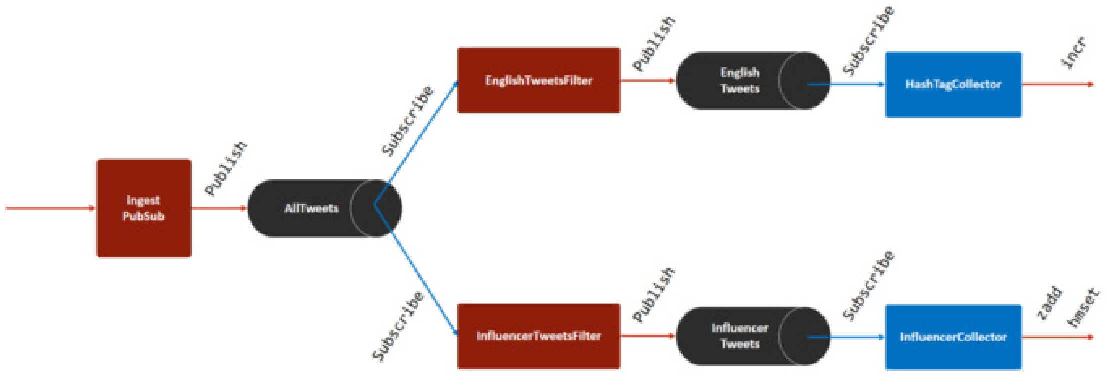

Figure 3: Redis publish/subscribe example [3]

streaming application could collect data from different types of sources such as temperature sensors and pressure sensors. In this case, each type of sensor would most likely send the data to its own respective stream for storage and processing. Each data entry in the stream can be a set of one or more key-value pairs, which could for example contain the sensor id, timestamp of sensor reading, and the sensor value itself.

For consuming the data, Redis Streams makes use of so-called consumer groups. One or more consumer groups can be attached to a single stream, where each group can contain one or more consumers. Consumer groups have two main purposes: distribution and scalability [16]. Having several consumer groups attached to a single stream means that each message in the stream will be delivered to every consumer group using a fan-out approach. This is useful in situations where the data is needed for several purposes. An application could for example have one consumer group responsible for saving the consumed data in a database, and another one for providing real-time analytics. For scaling the data processing, multiple consumers per consumer group can be used. In this case, each message in the stream is delivered to different consumers of the group, assuring that each message will be consumed only once. Using consumer groups with multiple consumers for scaling the processing is useful when dealing with high volumes of incoming data. Figure 4 showcases a simple streaming data pipeline featuring producers and consumers.

Working with the stream data structure in Redis is simple, and it only contains 13 different commands [21]. Examples of the most notable ones are as follows:

– XADD. Used to append a new entry to a stream using a unique identifier. If the stream does not yet exist when the first entry is added, the stream will automatically be created. An entry can be added as one or more key-value pairs.

– XREAD. Used to read entries from one or multiple streams at the same time. Able to read one or more entries at a time, while also specifying from what point in the stream (beginning, end, or starting from a specific entry ID). Can be used in a blocking or non-blocking fashion.

– XRANGE. Used to fetch entries in a specific time range by providing the start and end IDs. Often used to iterate a stream, for example when you want to go through past data. The opposing XREVRANGE command returns the entries in the reverse order.

– XGROUP. Used to manage the consumer groups of a stream; creating and attaching a group to a specific stream, deleting groups, as well as adding and removing consumers from a group.

– XREADGROUP. Similar to XREAD, but specifically for consumer groups. When a consumer has read the data, XACK should be used to inform the pending entry list that the data has been successfully read.

– XPENDING. Used to fetch entries that have not been acknowledged with XACK by a consumer that is part of a consumer group. Useful for example if a consumer disconnects and you want to process entries that have been left in the pending list.

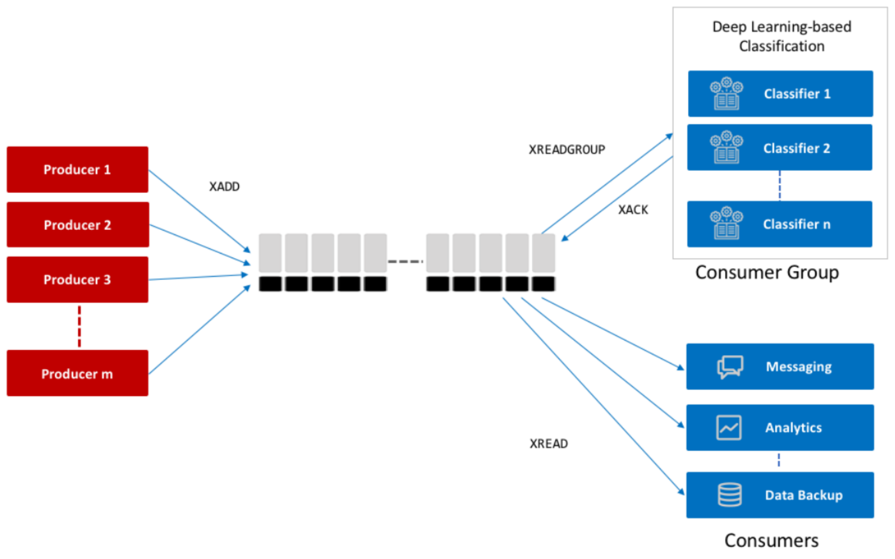

Figure 4: Redis Streams example [4]

2.5 Apache Kafka

Apache Kafka [22] is a distributed stream processing platform originally developed by LinkedIn. It was later open-sourced and donated to the Apache Software Foundation in 2011 [5], and has since become one of the more popular technologies used in stream processing pipelines. Kafka offers high performance messaging and processing due to its highly scalable architecture. At first, it was developed purely as a publish/subscribe messaging system, while simultaneously also offering durable on-disk storage that many other messaging systems did not. However, it has since transformed into a fully fledged stream processing platform. In Kafka version 0.10, the Kafka Streams DSL (domain-specific language) was introduced [23]. Kafka Streams introduced the possibility of also using Kafka as a stream processor for data transformation. A popular use case for Kafka has been to combine it with some other dedicated stream processing frameworks that do the actual processing, such as Apache Spark, Flink or Storm, while using Kafka simply as a high performance data ingestion and storage system. In the context of this thesis, we will focus on Kafka as a solution for this exact use case.

The central part of the Kafka architecture is the broker, which works as the intermediary between producers and consumers. It is responsible for ingesting incoming data from producers, storing the data on disk, as well as providing the data to consumers when requested. In a real production environment, it is always advised to run several Kafka brokers. This forms a so-called Kafka cluster, as showcased in figure 5. Each partition of a topic belongs to a single cluster, called the leader. Having a cluster setup with several brokers is not only a way to scale the system for incoming load, but it also provides a way to replicate the data for redundancy by assigning a partition to multiple brokers.

The data in Kafka is represented in a so-called commit log. Conceptually it works similarly to the stream structure is Redis, i.e. it stores the data captured from the data source in an append-only fashion. In Kafka, the terminology used for incoming events or data is a message, which consists of a key and value. There are two types of clients in Kafka; producers and consumers. Producers send messages to certain topics, while the consumers read the data by subscribing to these topics. Using topics allow for a logical separation of the incoming data. For example, data from different sensors would be sent to their respective topic. Topics are further divided into one or more partitions, where each partition is a single append-only log. By default, messages are evenly distributed over all partitions, but message keys can be used to assign messages to specific partitions. Having several partitions of the same topic provides not only redundancy, but also allows for scalability when multiple consumers of the same topic are used [5]. The word stream is often used to refer to the data belonging to a topic, going from the producer to the consumer.

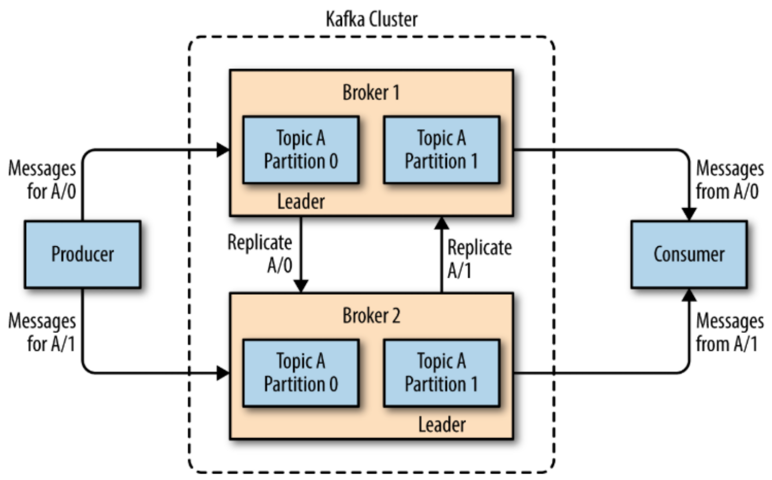

Figure 5: Kafka cluster with two brokers [5]

Like Redis Streams, the consumers of a Kafka topic are grouped into consumer groups, where a group can consist of one or more consumers. In Kafka, using consumer groups in conjunction with partitioned topics allows for scaling the data consumption in a horizontal way by adding more consumers to the group. Depending on the partition size of the topic, as well as the number of consumers that are part of the group, each consumer will be assigned ownership to one or more topic partitions. This assures that the data at each partition is only being consumed by one member of the group. Consumer groups also provide reliability by automatically rebalancing the members' partition ownerships in case of a consumer shutting down or losing connection. It is also possible to have several consumer groups attached to a single topic if for example multiple parts of a system need to consume the same data. In this case, the data can be consumed by each group independently of each other. Figure 6 describes an example consumer group interaction with a partitioned topic.

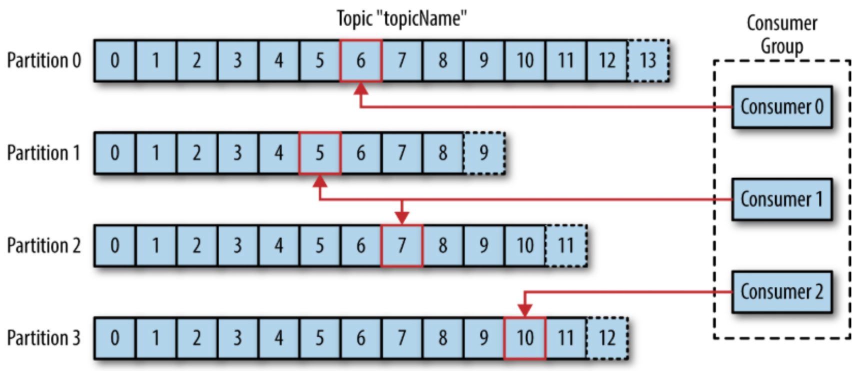

Figure 6: Kafka consumer group reading from topic partitions [5]

2.6 Apache Pulsar

Apache Pulsar [15] is an open source publish/subscribe messaging system built for high-performance distributed messaging [24]. While originally created by Yahoo, it has since become apart of the Apache Software Foundation. It is used for gathering and processing different events in near real-time, for use cases such as reporting, monitoring, marketing and advertising, personalization and fraud detection. For example, at eBay, Pulsar has been used to improve the user experience by analyzing user interactions and behaviors [25].

Pulsar is closely related to Apache Kafka in terms of features and use cases. It offers great scalability for message processing on a large scale, with high throughput and low end-to-end latency. Messages received are stored persistently with the help of Apache BookKeeper, and message delivery is guaranteed between producers and consumers [24]. While Pulsar is not a stream processing framework as the likes of Apache Storm or Spark Streaming, it does provide some light stream processing features with the use of Pulsar Functions [26].

A high-level overview of the architecture of a Pulsar deployment can be seen in figure 7. A Pulsar instance can consist of one or more clusters, where each cluster has the following components [27]:

– One or more brokers. The main function of the broker is to facilitate the communication between producers and consumers. It is also responsible for storing the incoming messages in BookKeeper instances (bookies) if persistent messaging is enabled. It is generally preferred to use several brokers in a single cluster for load balancing the incoming messages, as well as for availability in case of broker malfunction.

– A BookKeeper cluster that consists of one or more bookies. Apache BookKeeper is a storage system that is responsible for providing durable and persistent storage of messages. Number of bookies used in a cluster depends on the required capacity and throughput of the system.

– A ZooKeeper node for storing metadata, as well as responsible for cluster-level configuration and coordination. If the Pulsar instance consists of several clusters, a separate instance-level ZooKeeper node is also used to handle coordination tasks between clusters.

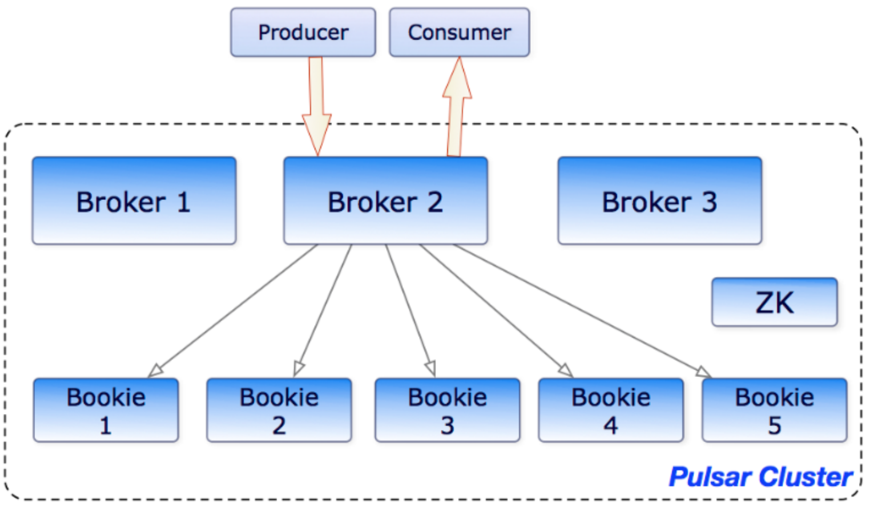

Figure 7: Overview of a Pulsar cluster [6]

Like Kafka, Pulsar is based on the publish/subscribe messaging pattern [28]. Producers send messages to certain topics, which are used to separate different types of messages. Consumers can then subscribe to specific topics to consume the data. The persistent storage that Pulsar offers means that all messages are retained, even when a consumer loses connection. The disconnected consumer can therefore easily reconnect and continue consuming the remaining data without any data loss.

Normally, a topic can only reside on a single broker. However, similar to Kafka, Pulsar offers partitioned topics that scale for higher throughput [28]. In contrast to normal topics, partitioned topics can be spread out over several brokers, taking full advantage of a multi-broker cluster. Figure 8 shows how the data is distributed from producer to consumer using a partitioned topic with five partitions (P0-P4).

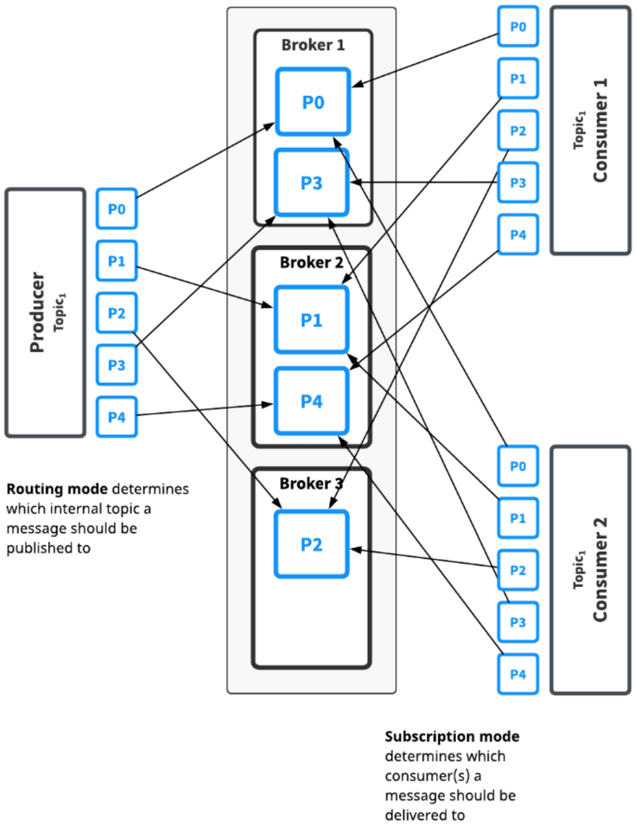
Producer
Topic₁
P0
P1
P2
P3
P4
Broker 1
Broker 2
Broker 3
Consumer 1
Consumer 2
Routing mode determines which internal topic a message should be published to
Subscription mode determines which consumer(s) a message should be delivered to

Figure 8: Subscription modes in Apache Pulsar [7]

Pulsar offers several different subscription modes for distributing the messages to consumers, as shown in figure 9. This includes the following modes:

- Exclusive. Only one consumer can be subscribed to the topic at a time.
- Failover. Several consumers can be subscribed to the topic at the same time using a master-slave approach. Only one of the consumers receive messages (the master). However, if the master consumer happens to disconnect, any subsequent messages will be directed to the following consumer (slave).
- Shared. Multiple consumers can be subscribed to the same topic. Messages are load balanced between all the connected consumers, i.e. messages are only consumed once. Shared subscription does not guarantee correct message ordering.
- Key shared. Similar to shared subscription mode, except that the message distribution is done based on key values.

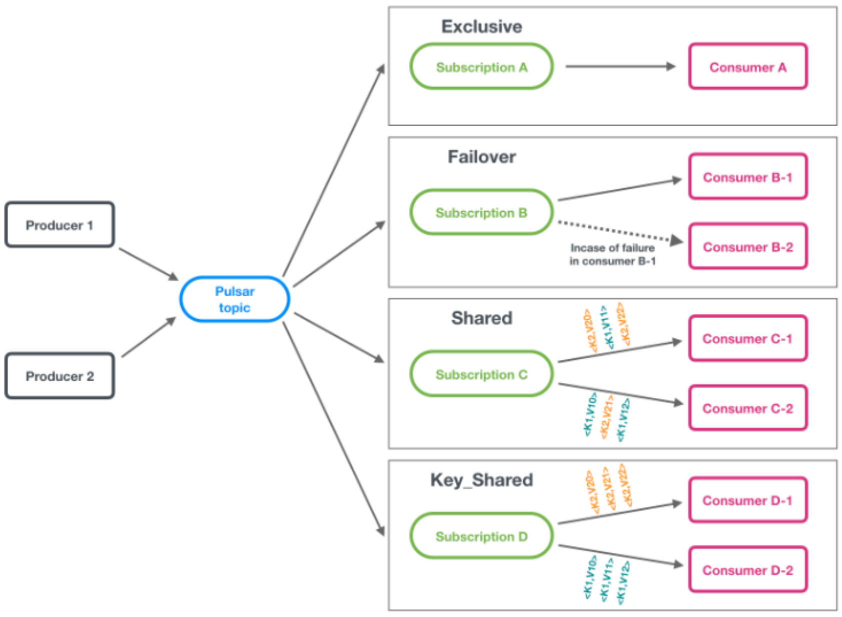

Figure 9: Overview of a Pulsar cluster [8]

3 Related Work

The purpose of the following section is to perform state-of-the-art. Stream processing related technologies that have been evaluated and compared in published research will be identified, as well as which metrics have been used for evaluation. It will also serve as a motivation for why the topic is worth researching based on the shortcomings mentioned in the papers. Only papers from the past couple of years have been considered due to the continuous advancements in the field, assuring that the results are relevant.

Comparative studies of stream processing technologies do exist in current research. Isah et al. [29] summarize the strengths and weaknesses of different distributed stream processing engines (DSPE) by performing a literature review of academia and industrial papers. Additionally, a subset of processing engines was chosen for a more in-depth comparison of certain key DSPE features such as the programming model, processing guarantees, state management, fault tolerance and more. According to the authors, the study revealed an apparent research gap when it comes to benchmarking different DSPEs based on performance and security.

Andreoni et al. [30] compared the performance of the open source stream processing platforms Apache Storm, Apache Flink and Spark Streaming. Throughput (messages processed per time unit) was used to evaluate performance, including how parallelism affects throughput. Additionally, the behavior under system failure was studied for the different platforms. The study also mentions the lack of performance evaluations and comparisons between different stream processing systems in research due to how recent the area of distributed stream processing is. Karimov et al. [31] also benchmark the performance of the same systems using throughput and latency as metrics.

The authors in [32] evaluate the performance of distributed stream processing frameworks in the context of Internet of Things (IoT) for building applications for Smart Cities. The stream processing frameworks that have been chosen for evaluation are Apache Storm, Apache Spark Streaming and Apache Flink, using throughput and latency as performance metrics. The conclusion drawn from the experiments done is that the appropriate framework to use depends on the requirements of the application being developed in terms of latency versus throughput. Apache Storm and Apache Flink are shown to have similar performance, while Spark Streaming offers higher throughput in exchange for a significant increase in processing latency.

In the paper [33], the performance of Apache Kafka as a data ingestion system is investigated in the context of Big Data streaming applications. The authors look at different configuration parameters and how they affect certain performance metrics to avoid bottlenecks for optimal performance. Configuration parameters such as message size, batch size, message replication and system hardware are looked at. Performance metrics such as throughput and latency, as well as CPU, disk, memory and network usage are used as evaluation criteria. The experiments show that there exists certain ranges of batch sizes that lead to better performance. Changing the amount of compute nodes used on the hardware side also lead to sudden performance drops in some cases. This behaviour is thought to be due to the synchronization in Kafka as well as the underlying network, but additional work must be done to verify this. The tests were run on a distributed testbed called Grid5000 [34] with up to 32 compute nodes.

The performance of the Apache Spark Streaming framework is evaluated in terms of throughput in [35]. The performance is evaluated with three different types of data source integrations; TCP, file streaming and Apache Kafka. The message sizes used range between 100 bytes to 10 megabytes, with CPU costs for processing each message ranging from 0 to 1 seconds. Spark Streaming with TCP integration achieved high throughput when the message size was small, with a rapid performance degradation as the message size was increased. Processing messages larger than 10^5 bytes proved to be unreliable no matter what frequency they were processed at. Spark with file streaming also performed best at lower message sizes and CPU loads. Integrating Spark Streaming with Apache Kafka performed well with message sizes smaller than 1 megabyte and with CPU loads under 0.1 seconds per message. The overall conclusion of the performance of the Spark Streaming framework is that it has excellent throughput when dealing with smaller messages and low processing costs, but struggles when processing larger messages in the 1 to 10 megabyte range.

Zeuch et al. [19] analyze the performance of different stream processing engines on modern hardware, using latency and throughput as metrics. They conclude that modern stream processing engines are not capable of fully taking advantage of modern hardware such as multi-core processors and high-speed networks.

Wu et al. [36] propose a model to predict the performance of an Apache Kafka setup in terms of throughput and end-to-end latency. The model uses the number of brokers, batch size and partition size of a topic as input parameters. The study concludes that the model shows great accuracy when compared to the experimental results, and that it will be developed further to take into account more configuration options.

In [37], the authors evaluate Apache Kafka on reliability by testing how Kafka handles message delivery under poor network conditions. Message loss rate and duplicate rate are used as metrics, and the results show that the size of data being sent matters the most under poor network conditions.

The authors in the paper [38] compare Apache Kafka and Apache Pulsar based on throughput and latency, as well as resource utilization in terms of CPU, RAM and bandwidth usage. The experiments show that Pulsar performs better in all aspects of performance. For small messages (1 KB) the maximum throughput of Pulsar is almost double the throughput if Kafka. Additionally, the average latency is about 20 times lower for Pulsar. For larger messages (1 MB) Pulsar still performed better than Kafka, however the gap was significantly decreased. Pulsar still managed a 10% higher throughput and a 10 time lower latency. The study concludes that even though Pulsar performs better than Kafka, it does not automatically make it the obvious choice. The authors mention that Kafka has a much richer ecosystem around it, with great integration to stream processor frameworks such as Apache Spark, as well as a much bigger community and user base in general.

According to the introduction to Redis Streams [16] on the official web site, Redis Streams is capable of handling 99.9% of the messages with a latency of under or equal to 2 milliseconds, with the rest falling between a 2 and 5 millisecond interval. The latency was calculated by attaching the current timestamp as an additional field to the message that was sent by the producer, which was then compared to the time the message was received by the consumer. Several producers were used to produce data at a rate of 10000 messages per second. The data was consumed by a single consumer group containing 10 consumers. The producers, consumers and the Redis server itself was running on the same dual-core instance used for testing, implying that the results would be better in a proper real-world setting. The size of the data sent per message was not specified, nor was the source code provided to allow for replication of the latency test.

Based on the papers reviewed, there does seem to be a need for high-quality performance evaluations and comparisons of different streaming platforms. While there does exist several Kafka related studies, newly emerged solutions such as Redis Streams can't be found in any published empirical studies. Studies related to Apache Pulsar are also almost non-existent. These factors make this a relevant area for conducting research.

4 Method

A high-level view of a real-time stream processing pipeline can be seen in *figure 10*. Producers send data (in this context, vehicle event data) to the data ingestion system from potentially various sources. The data ingestion system also works as a storage system, providing a means to store past events for durability and historical access. The data being ingested is then forwarded to the stream processor, which does the actual processing or transformation of the data. The processed data is then stored in some data warehouse for long-term storage and used for analysis. While the ingestion and processing part can be done using the same tool (e.g. Kafka with the Kafka Streams API), it is more common to have a separate dedicated stream processing framework connected to the data ingestion system. This thesis will focus specifically on evaluating technologies regarding the ingestion and storage part of the data pipeline, to sensibly narrow the scope of the thesis.

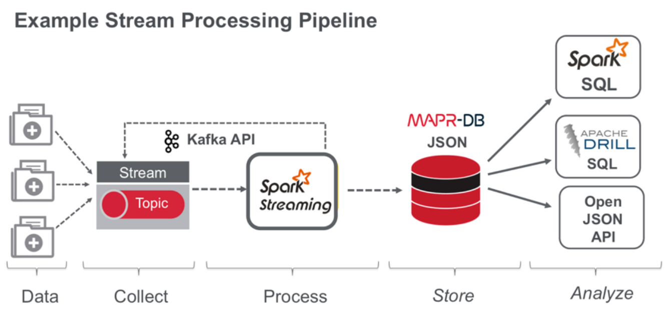

Figure 10: Example stream processing pipeline [9]

In order to address the research questions, two types of methodologies will be used; literature review and case study research. A literature review will be conducted to perform state-of-the-art. This will give an overview of several things, including identifying relevant streaming platforms for evaluation, analyzing benchmark results currently found in research, as well as which evaluation criteria have been used for comparing these types of technologies in terms of characteristics and performance metrics. The state-of-the-art will also provide the motivation for why the topic is worth researching from an academic point of view. Additionally, it will provide the necessary background information for the reader to understand what is being studied, i.e. fundamental terminology and concepts. The literature review will answer *RQ1* by reviewing current published research, white papers, books and official documentation for the chosen technologies.

To answer *RQ2*, a case study will be conducted. Case study research is an appropriate method when wanting a deeper understanding of the phenomena being studied in its real-world context; the industrial context here being the software development project at the company. It is also a proven methodology in software engineering, with guidelines on how to conduct and report the research. The guidelines provided by *Runeson et al.* [39] will be used and adapted for this specific context. The overall workflow looks as follows:

1. Identify a set of technologies for evaluation and comparison. This decision is done based on the state-of-the-art performed as well as what the company wants.
2. Give an overview of the chosen technologies.
3. Create a test environment for each of the chosen technologies.
4. Benchmark each of the technologies using the chosen performance metrics (latency and throughput) and analyze the data.
5. Report and discuss the results.

4.1 Company context

The thesis work is done in collaboration with the company Addiva AB [17]. They are involved in both hardware and software, however the work conducted during this thesis is entirely software based. One of their provided software solutions is called AddTrack [40], as showcased in *figure 11*. AddTrack provides the possibility for collecting and storing vehicle data, with an easy to use graphical user interface for analysis of both historical and real-time data. The software reports errors in the vehicles by detecting anomalies in the captured data. Some component of the vehicle might not be working correctly, for example the doors of a train. One of their customers is Bombardier [41], who use the software in the trains they manufacture.

The software currently runs on the old and deprecated Silverlight application framework, which is ending support in 2021. They are also not taking advantage of proper stream processing technologies for the real-time data processing pipeline. Due to these reasons, they are currently in the process of rearchitecting the software from scratch, and thus interested in selecting the proper technologies for solving the real-time data processing pipeline.

As the software is being built from scratch, there is no concrete system under test as such. Therefore, a test environment will be setup to benchmark the performance (maximum throughput and end-to-end latency) for each of the technologies. Due to confidentiality reasons, real customer data will not be accessible. However, for performing the benchmarks, that is not necessary. When benchmarking the performance, there are two main parameters that affect the results; size of the message in bytes and the rate at which messages are produced per time unit. The tests will be carried out using dummy data that adheres to their data specification (i.e. dummy data that is similar in size to the real data). The rate of messages sent per time unit for the tests is also determined by the company to reflect their needs.

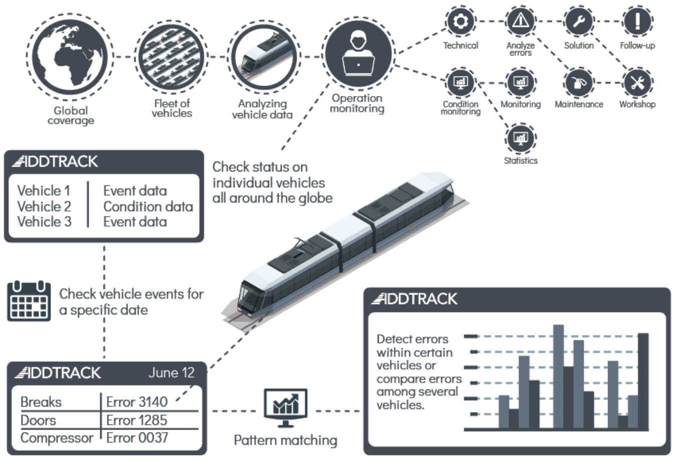

Figure 11: AddTrack - Vehicle analysis software [10]

4.2 Evaluation criteria

The chosen technologies will be evaluated on two fronts; *characteristics* and *performance*. The different types of characteristics of the system that will be evaluated are as follows:

– *Data persistence and retention.* How does the platform manage data persistence, where does it store the data, and for how long is the data accessible to consumers.

– *Fault tolerance.* How does the platform provide means to make sure that complete system failure can be mitigated.

– *Language and container support.* For which languages do client libraries exist (official versus third party libraries), and what are the differences in terms of feature support. The company also intends to run everything in a containerized environment, so whether there exists ready-made Docker images is of interest.

For evaluating performance, the following metrics will be used:

– *Throughput.* The maximum load that the platform can handle. Measured in terms of messages/second and MB/second.

– *End-to-end latency.* The amount of time it takes for messages to travel between a producer and consumer. Mean, minimum and maximum latency values are measured in terms of milliseconds.

4.3 Benchmarking set-up and configuration

Since the company is interested in deploying their application in a containerized environment, all the platforms under performance evaluation have been setup as such. Docker containers have been used, and the Dockerfiles and respective commands for running them can be found in the appendices.

The performance benchmarks for measuring throughput and latency will be executed on a single machine setup. The machine is deployed on the cloud service provider UpCloud, running Debian 10 (Buster) as an operating system. Regarding the hardware, it is important that the machine running the benchmarks is powerful enough to handle realistic workloads. For example, running the tests on a basic quad-core machine configuration would not yield interesting results, as this type of setup would most likely never be used in a real-world use case. Generally, it is recommended that these type of data ingestion systems should run on CPUs with a high core count, as well as enough memory and disk space. The relevant hardware configurations used for the benchmarks are as follows:

– 20 Core Intel(R) Xeon(R) Gold 6136 CPU @ 3.00GHz

– 32GB RAM

– 200GB SSD storage

In a production environment, it is usually recommended to run a multi-node cluster (i.e. each node on a separate machine) not only for scaling the system to the incoming load, but most importantly for assuring high availability by reducing the possibility of data loss and providing reliable service. This stands true for both Redis, Kafka and Pulsar deployments. However, in this context, all the platforms under test will be running on a single machine. The reasoning for this is that when testing the end-to-end latency (latency between producer and consumer), we are only interested in the processing latency caused by the platform itself. If the producers, consumers, and brokers were to be run on separate machines on the cloud, there would be additional network latency caused by the communication between the machines. By running the tests on a single machine setup, this additional overhead will not affect the results.

The platform specific configuration file settings used are the default values provided with the installation, i.e. an out-of-box setup. Redis, Kafka and Pulsar all have extensive configuration options that can be fine tuned for specific message sizes, throughput rates, and underlying hardware. However, figuring out the most optimal platform specific configuration parameters is simply out of scope for this thesis. All platforms are tested with persistent storage enabled, i.e. incoming messages are saved on disk. This has a significant effect on the performance, but it also represents most real-world use cases, including Addiva's intended use.

4.4 Message sizes and throughput rates

The platforms are evaluated using three different message sizes: 1 kB, 65 kB and 600 kB. These message sizes are based on the amount of data sent by the different vehicle models, only taking into consideration the worst-case scenario. For measuring throughput, the systems are stress tested to the max to find out the highest possible throughput rate (messages/sec and MB/sec), i.e. no throttling is done. The built-in platform specific benchmarking tools will be used when evaluating the maximum throughput.

When measuring the end-to-end latency for each of the message sizes, a fixed throughput rate is used. Since each of the platforms can handle different throughput rates for each message size, it is important that the end-to-end latency is evaluated at a fixed message rate that can be handled by all the platforms so that the results can be fairly compared. Each message size has been tested with two different message rates. These are as follows:

– 1 kB message size at 1000 and 5000 messages/sec (0.95 MB/sec and 4.76 MB/sec)

– 65 kB message size at 100 and 500 messages/sec (6.19 MB/sec and 30.99 MB/sec)

– 600 kB message size at 50 and 200 messages/sec (28.61 MB/sec and 144.44 MB/sec)

These latency tests are run using two consumers as part of a group that read incoming data, and a single producer that sends data at the constant rates mentioned above. The Kafka and Pulsar topics used are configured with two partitions, one for each consumer. The measured values are the minimum, maximum and average latency between the producer and consumer. The most relevant measurement is the mean value, as it is the best overall latency performance indication of a platform.

The latency tests for each platform have been written using Python 3.7.3 and the following client libraries:

– Redis-py (version 3.5.2) for Redis 6.0.2

– Pulsar-client (version 2.5.1) for Apache Pulsar 2.5.1

– Confluent-kafka (version 1.4.1) for Kafka 2.5.0

How to reproduce the throughput and latency tests, as well as setting up the respective platform environments in a containerized environment using Docker can be seen in the appendices.

4.5 Ethical and societal considerations

No ethical or societal considerations have to be addressed in order to conduct and report the research. As we do not have direct access to the customer data due to privacy concerns, dummy data that conform to the type description of the event data will be used for the performance benchmarks.

5 Results

5.1 Characteristics comparison

This section will present the results of the characteristics comparison between Kafka, Pulsar and Redis. Overview of the results can be seen in Table 1.

5.1.1 Data persistence and retention

While Redis stores data in memory, it also provides data persistence on disk [42]. Redis can be configured for several persistence options; RDB persistence, AOF persistence, a combination of both, or no persistence at all. RDB (Redis Database Backup) is the default persistence mode in Redis. It creates a point-in-time snapshot of the current data set in a single binary *.rdb* file. Redis can be configured to create a snapshot based on a specified time interval or when a specific amount of writes have been done. Using RDB persistence is great for backups and disaster recovery, however it is not suitable if minimizing data loss is a priority. For example, if a sudden power outage was to happen, the data written after the latest snapshot would be lost. AOF (append-only file) persistence is a much more durable approach, as it stores every write operation in an append-only log file. This file can then be used to recreate the data set without data loss if a potential power outage or some other disruption were to happen. An AOF file is bigger in size than a RDB snapshot, and can be slower than RDB depending on the fsync policy that is used. As of right now, the only reliable way of managing the data retention of a stream is to use the *MAXLEN* argument when adding entries to the stream with *XADD*. This provides a means to limit the length of the stream. When the max limit is exceeded the stream will be trimmed, making place for newer entries. There is currently no option to automatically remove elements from the stream based on for example size (in terms of bytes) or time limits.

In Kafka, data persistence is done on disk [43]. All the data is written to a persistent log file that can be replicated and distributed across multiple machines for redundancy. Kafka's role as a storage system is very dependent on the file system and page caching for providing persistent messaging with great performance. The data is written directly to the log on the file system without necessarily flushing to disk, instead of storing as much data in memory as possible until a flush to disk is forced due to running out of memory space. This is seen as a superior design choice compared to maintaining an in-memory cache or some other mechanism. Messages in Kafka stay in the persistent log even after having being consumed. The retention policy for how long messages are kept in the log can be configured based on a specific period of time, or when the log has reached a certain size. Retention policies are configured on a per-topic basis. Messages are deleted regardless whether they have been consumed or not when the retention policy is surpassed in time or size. Due to the design of Kafka, having long retention periods, i.e. several days or weeks, is not an issue from a performance point of view. There is no way to directly turn off persistent storage due to the core architectural design of Kafka, however the retention period can be set to a small time period or file size to delete the data at frequent intervals.

In Pulsar, the data is also persisted on disk [27]. However, in comparison to Kafka, Pulsar relies on a separate storage system called Apache BookKeeper for providing durable and persistent messaging. BookKeeper is responsible for handling the ledger(s) of a specific topic, which is an append-only data structure that stores the data, similar to the commit log in Kafka. It is common to have multiple BookKeeper nodes in a Pulsar cluster, where each ledger is assigned to multiple nodes. This is done for both performance and redundancy reasons. By default, Pulsar deletes messages once they have been consumed and acknowledged by all attached subscribers [44]. This default behaviour can be overwritten by setting retention policies. Similar to Kafka, retention policies can be set based on a specific size or time limit. Additionally, Pulsar topics can also be configured with a separate time-to-live (TTL) property. This property can be configured to delete messages that have not been acknowledged by any consumer within a specific time frame. In Pulsar it is also possible to directly create non-persistent topics that do not save messages on disk, but only keep them in memory. This may lead to data loss in case of a broker failure or a subscriber disconnecting. One can also expect better performance in terms of throughput and latency when using non-persistent topics, as no communication between brokers and BookKeeper nodes is required.

5.1.2 Fault tolerance

At a basic level, Redis supports a master-slave type of replication for providing availability [45]. The master Redis instance is responsible for updating the slaves when some change has happened in the master node. If a slave node disconnects from the master node for example due to network issues, it will automatically try to reconnect and perform partial or full resynchronization. Partial synchronization means that the slave node will try to obtain the stream of missed commands during the time it was disconnected and replicate them. In case a full resynchronization is required, the master node will need to take a snapshot of all its data, which can then be used to recreate the data set in the slave node. Tools such as Redis Cluster[46] and Redis Sentinel[47] also exist, which offer even more high availability features. These should preferably be used when running Redis at scale.

In Kafka, a fault tolerant deployment means having multiple brokers and ZooKeeper nodes on separate machines in the cluster. The concept of partitions in a topic is what drives the replication in Kafka [43]. Each partition has a leader, with zero or more followers. The number of followers, including the leader, is referred as the replication factor of a topic. This replication factor is set on a topic-by-topic basis. The highest possible replication factor is the number of brokers in the cluster, i.e. a topic with a replication factor of five requires five brokers in the cluster. If one of the servers in the cluster is shut down due to some failure, automatic failover will be done, assuring that no messages will be lost and the cluster can continue providing reliable and correct service.

Fault tolerance in Pulsar is similar to Kafka, i.e. having a cluster with multiple brokers, ZooKeeper nodes and BookKeeper nodes across several machines. In Pulsar, the ledgers can be replicated to multiple BookKeeper nodes, making sure that data can be continuously stored even if one of the BookKeeper nodes die [27]. Pulsar also has built-in support for geo-replication [48]. Geo-replication allows for replication of messages across several clusters, where each cluster might reside in different geographical locations. This is a means to provide disaster recovery capabilities for Pulsar.

5.1.3 Language and container support

Redis is supported by most programming languages, including Python, Java, C/C++, C#, Nodejs, and many more [49]. Several different clients exist for each language with various feature support. As Redis Streams was introduced in Redis 5.0, there are still many clients that do not support interacting with the streams data structure. Therefore, if one is tied to developing in a specific language, it might be the case that there does not exist a client library for that language that supports the streams data structure. Redis also offer an official Docker image for deploying a Redis server in a containerized environment [50]. The documentation however is very basic, with no mention of how to handle production level deployments with several nodes for redundancy, fault tolerance etc. in a containerized environment. Another popular Docker image is by Bitnami [51], which offers far more detailed information about these things. Additionally, they also offer Helm charts for deploying with Kubernetes [52], a popular container orchestration tool.

The official client language for interacting with Kafka is Java, which is also the language that Kafka itself is written in. Much like Redis, Kafka also has third-party client support for most of the popular languages like Python, C/C++, C#, Golang, Nodejs and more [53]. The main difference between the main java client and the third-party ones is feature support. For example, most third-party client only support the basic consumer and producer APIs, i.e. creating clients for sending and receiving data directly to and from Kafka. While it might be enough for most cases, there might be situations where you want to pull data from an external data source, such as a database, data warehouse, or some other application, using the Connect API. The official java client is also the only one that supports Kafka Streams, which is the library for taking advantage of Kafka's stream processing capabilities. If Kafka is expected to be used both as a data ingestion system and as a stream processor, then using the official java client is a requirement. There is no official Docker image maintained by the Kafka team, however two popular alternatives are by Bitnami [54] and Wurstmeister [55]. The Bitnami image offers far more detailed documentation regarding deployment, running multiple brokers, security and important configuration settings. The Docker image itself is also much more regularly updated with bug fixes and recent features compared to the Wurstmeister image. Like all Bitnami images, they also offer Helm charts for deploying in Kubernetes.

For Apache Pulsar, officially released client libraries can be found for Java, Golang, Python, C++ and Nodejs [56]. Pulsar can also be communicated with over the WebSocket protocol, meaning that any programming language that has a WebSocket client can connect to Pulsar. Communicating with Pulsar through the WebSocket protocol gives access to the basic consumer and producer features, but lacks some of the more advances features that the native client libraries offer. For example, features such as consuming multiple topics at the same time, key-based subscription mode, support for data schema, and more, are not supported over WebSockets [57]. However, for basic use, this is a flexible way to work with Pulsar in languages that do not have an officially supported client. For deploying Pulsar in a containerized environment, there exists an officially maintained Docker image by the Pulsar team. This image is however only for starting Pulsar in standalone mode, i.e. running a single broker, ZooKeeper and BookKeeper node. For deploying with multiple nodes in Docker with Docker Compose for example, one would need to create separate custom images for the different parts (broker, ZooKeeper, BookKeeper) which could then be scaled accordingly. They also offer an official Helm chart for running Pulsar in kubernetes.

Table 1: Summary of the characteristics comparison between Kafka, Pulsar and Redis.

	Kafka	Pulsar	Redis
Data persistence	Data is stored on-disk. Relies heavily on filesystem and page caching for providing persistent messaging. Easy to scale storage capacity by adding more disk space.	Data is stored on-disk. Relies on an external storage system called Apache BookKeeper for providing persistent messaging. Easy to scale storage capacity by adding more disk space.	Data is stored in-memory. How much data that can be persisted is limited by the amount of RAM. Provides separate on-disk persistence modes, but mainly used for backups and disaster recovery.
Data retention	Data retention policies based on specified size and time limits. Older messages removed after storage exceeds size limit or certain amount of time has passed.	Same data retention policies as Kafka.	Redis Streams does not have support for any data retention policies. Have to manually trim the size of the stream to remove old messages.
Fault tolerance	Achieve fault tolerance by scaling the nodes in the cluster, i.e. adding more brokers and ZooKeeper nodes. Each topic can be given a replication factor for redundancy.	Same as Kafka. Additionally, more BookKeeper nodes for handling persistence should be added. Built-in support for geo-replication, allowing replication across several clusters.	Master-slave principle. Write commands to the master nodes are also replicated on the slave nodes. Support for additional tools such as Redis Cluster and Redis Sentinel for even higher availability.
Language support	Offical client in Java, but third-party ones in most popular languages. Must use the offical Java client to get exposure to all the Kafka APIs.	Official clients in Java, Python, C++, Golang and Nodejs. Can also be interacted with via the WebSocket protocol. Third-party clients for most languages.	Several clients for most programming languages. However, a limited amount of clients that have implemented support for the stream data structure.
Container support	No officially maintained Docker image. Several popular third-party maintained ones.	Officially maintained Docker image for running Pulsar in standalone mode (single node cluster). Have to create own images for deploying a multi-node cluster.	Officially maintained Docker image. Also popular third-party images.

5.2 Performance comparison

This section will present the results of the throughput and end-to-end latency results. An overview of the throughput results are presented in Table 2.

5.2.1 Throughput

The overall results show a rather major difference in maximum throughput rates for different message sizes when comparing platforms, as shown in figures 12 - 14. When a payload of 1 kB is used, Pulsar performs the best with 195 068 messages/sec (186 MB/sec). Kafka manages to handle 137 676 messages/sec (131.30 MB/sec), significantly behind Pulsar by almost 35%. Redis performs by far the worst with a throughput rate of only 54 124 messages/sec (51.61 MB/sec). This is about two and a half times lower than Kafka's rate, and almost four times lower than what Pulsar is capable of handling.

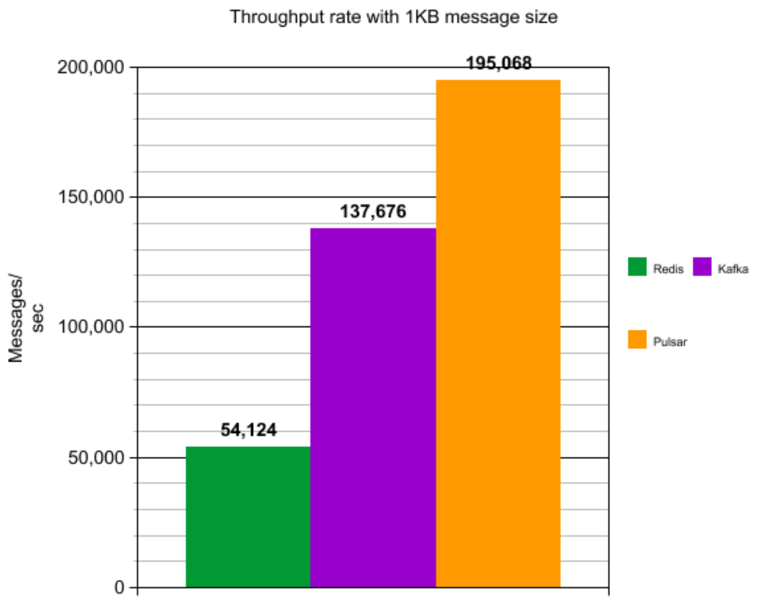

Figure 12: Maximum throughput measured in messages/sec with a message size of 1 kB.

When significantly increasing the message size from 1 kB to 65 kB, Redis becomes a much more competitive choice with a throughput rate of 4684 messages/sec (290.35 MB/sec). This is almost identical to Kafka, which manages to handle 4738 messages/sec (293.70 MB/sec). Running the test for Pulsar, however, is when potential problems start to arise. The built-in producer performance benchmark requires the rate of messages as an input. When testing for maximum possible throughput, this parameter should be set very high to stress test the system to the max. However, when running the benchmark with a message rate of for example 5000 or higher, the results were very inconsistent and low in performance, resulting in an average of 885 messages/sec (54.86 MB/sec). After some testing, limiting the message rate to 3000 gave the best consistent output, resulting in 2953 messages/sec (183.05 MB/sec). This is still quite a bit worse than both Kafka and Redis, with a difference in around 110 MB/sec maximum throughput rate. Pulsar went from the best performing platform to the worst performing one when increasing the message size to 65 kB. The results for Pulsar seem to indicate some type of bottleneck in the cluster when reaching a specific throughput rate of around 180-185 MB/sec. The significant increase in message size could also be an affecting factor.

Running the benchmarks with an even greater message size of 600 kB shows the struggle of both Redis and Pulsar. Redis manages a total of 337 messages/sec (192.83 MB/sec), which is roughly 100 MB/sec lower than what it was capable of with a message size of 65 kB. It seems to indicate that Redis is having issues dealing with very large message sizes. Usually, the higher the message size is, the higher the throughput in terms of MB/sec (but lower messages/sec). This is the case

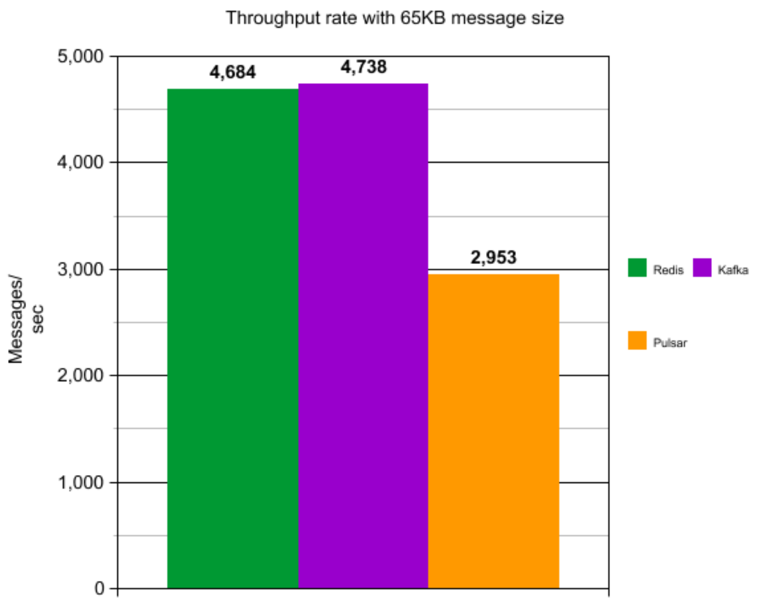

Figure 13: Maximum throughput measured in messages/sec with a message size of 65 kB.

with Kafka, which performs very well at 870 messages per second (497.81 MB/sec). For Pulsar, we run into the same issue as in the previous test when trying to run the 600 kB benchmark at a high message rate. However, this time it resulted in an out of direct memory error which ended up killing the BookKeeper node. The implication here is that a single BookKeeper node is not able to handle very high throughput levels in terms of MB/sec, causing it to be the bottleneck of the cluster. For example, running the same benchmark on a non-persistent topic in Pulsar (no storage, i.e. no interaction with BookKeeper) results in an impressive 2764 messages/sec (1581.57 MB/sec) maximum throughput. These non-persistence results show that the broker is more than capable of handling a lot data, but a single BookKeeper node is not enough when persistence is required at high levels of MB/sec. This behaviour seems to be verified by what another user has experienced in [58]. For stable results, the 600 kB benchmark had to be run at a max message rate of 300, which resulted in an average of 293 messages/sec (167.65 MB/sec). This performance is on par with Redis, however both Redis and Pulsar perform poorly compared to Kafka in this scenario.

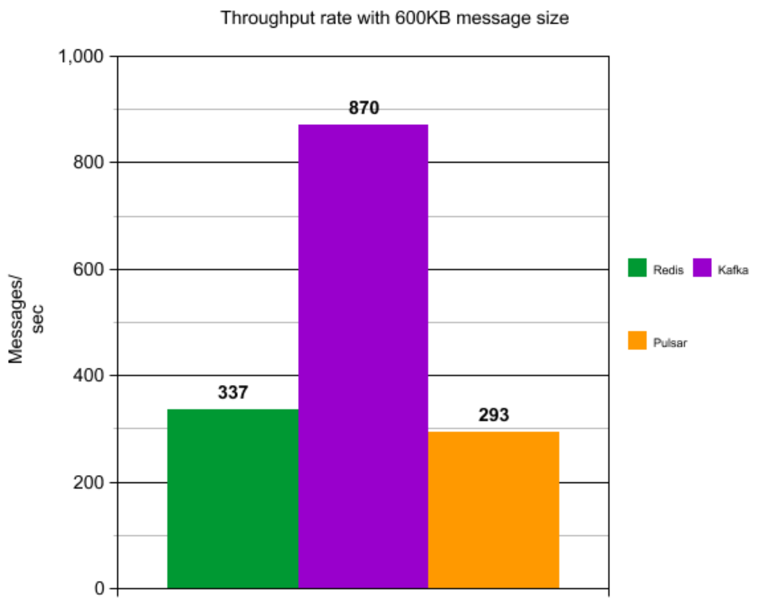

Figure 14: Maximum throughput measured in messages/sec with a message size of 600 kB.

Table 2: Summary of the throughput results measured in messages/sec and MB/s for message sizes 1 kB, 65 kB and 600 kB.

	Message Size					
	1 kB		65 kB		600 kB	
	Msg/s	MB/s	Msg/s	MB/s	Msg/s	MB/s
Kafka	137 676	131.30	4738	293.70	870	497.81
Pulsar	195 068	186	2953	183.05	293	167.65
Redis	54 124	51.61	4684	290.35	337	192.83

5.2.2 End-to-end latency

All the end-to-end latency results are reported in Table 3-8 based on the parameters described in section 4.2.

Redis and Pulsar perform the best in terms of average latency for the smallest message size of 1 kB, as seen in Table 3-4. Both platforms manage a sub millisecond average latency with 1000 and 5000 messages/sec. Kafka is notably worse, with an average latency that is 3-4 times higher than both Redis and Pulsar. Kafka also has a significantly higher max latency when running at a rate of 5000 messages/sec.

Table 3: Latency in milliseconds with message size 1 kB and message rate 1000 msg/sec.

	mean	min	max
Kafka	1.714	0.406	5.312
Pulsar	0.557	0.300	5.919
Redis	0.440	0.175	4.013

Table 4: Latency in milliseconds with message size 1 kB and message rate 5000 msg/sec.

	mean	min	max
Kafka	1.995	0.337	21.627
Pulsar	0.472	0.168	6.102
Redis	0.660	0.183	3.395

Sending larger messages (65 kB) at 100 and 500 messages/sec still shows that Redis and Pulsar are performing the best in terms of average latency, as shown in Table 5-6. Running at 100 messages/sec (6.19 MB/sec), Redis is the fastest with Pulsar slightly behind, with both being roughly twice as fast as Kafka. Interestingly, both Redis and Pulsar perform better at an increased message rate of 500 (30.99 MB/sec), resulting in an average latency 4-5 times lower than what Kafka can handle. For Kafka the latency gets worse as the overall throughput increases, which is the behaviour one would expect. The increased throughput also results in a notable increase in max latency for Redis and Kafka.

Table 5: Latency in milliseconds with message size 65 kB and message rate 100 msg/sec.

	mean	min	max
Kafka	2.842	1.991	5.532
Pulsar	1.534	1.029	8.064
Redis	1.237	0.628	12.291

Table 6: Latency in milliseconds with message size 65 kB and message rate 500 msg/sec.

	mean	min	max
Kafka	3.499	1.572	44.027
Pulsar	0.704	0.426	5.530
Redis	0.975	0.456	41.219

When getting into the largest message size of 600 kB and higher throughput rates in terms of the total amount of data being sent, we start seeing some interesting changes, as seen in Table 7-8. For example, both Redis and Pulsar performs significantly worse running at 50 messages/sec with a message size of 600 kB (28.61 MB/sec) compared to running at 500 messages/sec with a size of 65 kB (30.99 MB/sec). The resulting average latency is approximately 4-5 times higher, even though the overall throughput rate in terms of MB/sec is lower. This shows the negative effect that message size has on average end-to-end latency. Interestingly, the average latency for Redis and Kafka is roughly the same running at 50 messages/sec (28.61 MB/sec) as compared to running at 200 messages/sec (144.44 MB/sec), even though the overall throughput is about 5 times higher. The large message size also introduces very significant increases in max latency for both Redis and Kafka (especially Redis) compared to Pulsar. Pulsar once again proves to be the best performing platform in all the latency aspects.

Table 7: Latency in milliseconds with message size 600 kB and message rate 50 msg/sec.

	mean	min	max
Kafka	5.407	3.770	65.192
Pulsar	2.782	2.231	6.114
Redis	5.350	3.352	45.216

Table 8: Latency in milliseconds with message size 600 kB and message rate 200 msg/sec.

	mean	min	max
Kafka	5.728	3.502	71.531
Pulsar	3.773	2.564	12.949
Redis	5.357	3.378	154.397

6 Discussion

Comparing the platforms based on their features as a storage system we can see some clear differences in how they operate. While all the platforms offer persistent storage, the way they do it and to what capacity is quite different. Kafka and Pulsar are both similar in that they write the data directly to disk, the only difference being that Pulsar relies on an external storage system to do so (Apache BookKeeper). This data can be retained for both short and long periods of time and is mostly only limited by the amount of available disk space. This makes it extremely easy to for example consume messages from several weeks ago. Redis on the other hand works in an entirely different way. In Redis, the whole dataset lives in main memory. Any data that goes into a Redis stream, or any Redis structure for that matter, does not get written directly to disk. With AOF persistence enabled, Redis only writes all the write operations that were used in a file on disk, which can then be used to recreate the dataset in memory when needed. The RDB persistence option can also be used, but it is mostly for backups and disaster recovery.

The solution to which platform to use based on persistence requirements is entirely dependent on the use case. If long term historical access is required, then Kafka and Pulsar are the obvious choice. They are both designed as durable messaging systems able to handle and persist large amounts of data. Redis on the other hand is limited by main memory, which dictates how large of a dataset is readily available. This makes Redis a potential choice for short-term storage persistence requirements. Alternatively, a streaming pipeline could always combine both Redis and Kafka/Pulsar for different types of data streams. However, considering that both Kafka and Pulsar essentially cover the same use cases as Redis, it is questionable whether that is worthwhile. Maintaining several platforms would also increase the overall complexity of the streaming pipeline, including the extra maintenance that comes with it.

Another important aspect to consider is the maturity of the technology. These modern data streaming solutions are very much considered new technologies, as they have come up in the spotlight during the past couple of years. While Redis has been around for over a decade by now, the streams structure for building real-time data pipelines was only introduced a couple of years ago in version 4.0. Apart from the official documentation and a couple of blog posts, it does not seem to be something that has gotten proper industry attention. In the case of Kafka and Pulsar, they were both developed and used by large companies before being open sourced to the public. While Pulsar is currently being used in industry at a large scale, it is in a lot of ways overshadowed by the popularity of Kafka. They are both remarkably similar in terms of architectural design and the problems they solve. This brings up the question of why it is that Kafka has received the largest userbase out of the three. One probable reason is the overall ecosystem surrounding it. Although this study is specifically focused on evaluating and comparing systems for the data ingestion part of the stream processing pipeline, it is important to look at the bigger picture. In a lot of cases you might want to connect the data ingestion system with a dedicated stream processing framework if more complex processing is required. In the case of Kafka, Kafka itself can be used as a stream processor with the Kafka Streams framework. However, more importantly, Kafka has great compatibility with most of the popular stream processing frameworks. Apache Storm, Flink and Spark for example all have built-in support for Kafka as a data source. This type of support and flexibility is something that Pulsar and Redis Streams simply do not offer and is most definitely something that should be considered when choosing platform.

To expand on this, language support for these platforms is another quite important factor, as it is something that might affect the overall development time and maintenance of the project. Kafka, Pulsar and Redis all have client libraries for most of the popular languages. However, a lot of these clients are third-party developed and maintained, meaning that they often lack the features of the native clients. In the case of Kafka, most client only support the basic consumer and producer API. For getting access to all the features, including the Kafka Streams API for using Kafka as an actual stream processor, the native Java client must be used. Pulsar offers slightly more variety in official clients that have full feature support and is in general quite language agnostic considering the support for the WebSocket protocol. Redis in general supports a wide variety of languages with many different client libraries, but a lot of them have not implemented support for Redis Streams. However, for most languages there does exist at least one client that supports the stream datatype.

The other important aspect is related to performance. If the requirement is to build real-time data pipelines that can handle large amounts of incoming data at low latency, then you also need a platform that is able to achieve this. In this case the platforms were evaluated on throughput and latency. These metrics have been widely used in the related works, as they best describe the raw performance needs. The performance tests in this study do not take into consideration other metrics such as CPU and RAM utilization, which was also measured for example in [38] and [33]. This might be of interest when for example optimizing the hardware used for running these systems.

In terms of maximum throughput, Kafka is the most consistent one across all the tested message sizes. Pulsar performs very well with small messages, however the bottleneck of running a single BookKeeper node for persistent storage severely affects the potential performance at higher throughput rates. Considering that Pulsar was able to handle data at 1.5 GB/sec on non-persistent topics as explained in section 6.2.1, the results would be widely different in a multi-node setup with several BookKeeper nodes for handling persistence. That being said, you also cannot compare the results of persistent versus non-persistent messaging speeds, as the overhead of writing the data to disk is quite significant. The tests that were conducted in [38] show that Pulsar outperforms Kafka in raw throughput for both 1 kB and 1 MB message sizes. While those tests were run in multi-node cluster, it shows that Pulsar is capable of outperforming Kafka when there is no bottleneck related to not having enough BookKeeper nodes for handling persistent storage. The performance of Redis is quite interesting when looking at the results, as it seems to not be able to handle too many small messages per time unit, nor does it perform all that great with large 600 kB messages. Interestingly though, it performs similarly to Kafka with a message size of 65 kB. It should however also be noted that these throughput benchmarks test the absolute maximum limit of the system. This will always result in a very notable increase in latency since there will inevitably be some longer queuing or buffering done. In that respect, expecting to run the system at these throughput levels might not be practical at all if latency is a concern.

In most applications, having the lowest possible average latency is the best measurement of performance. In this regard Pulsar is the best performing platform across all the tested message sizes and throughput rates. While there is quite a large relative difference in average latencies between Pulsar and Kafka, the absolute difference is just a couple of milliseconds. The latency comparisons in [38] show a much larger difference in latencies between Kafka and Pulsar, but they were also tested at widely different throughput levels. Both Redis and Pulsar are extremely fast at low throughput and message size, resulting in an impressive sub millisecond latency. However, Redis is also the platform that gets most affected by the largest 600 kB message size, both in average and max latency. The maximum latency of 150 ms when running at 144 MB/sec is quite significant. There might be certain applications where this is particularly important, i.e. not being able to process the message within a specific maximum time frame negatively affects the output.

Choosing a platform based on performance requirements generally comes down to a couple of things. Firstly, throughput requirements must be met. This includes being able to easily scale up the system to handle increasing workloads, which is something that can be done in Redis, Kafka and Pulsar by adding more nodes to the cluster. Secondly, the latency requirements must be met. When talking about real-time streaming applications, it usually means latency measured in milliseconds. How strict the latency requirements are depends on the domain and type of application. In general, the decision is often a compromise between high throughput and low latency, as mentioned in [32]. Stressing the system to the very max will always have a negative effect on latency. However, for some applications low latency might not be that important compared to having a high throughput.

6.1 Limitations

The main limitations of the study concern the results of the performance evaluation. The tests that have been performed are very much context dependent. These platforms have been evaluated based on specific hardware, message sizes, throughput levels and cluster size. While the results are reproducible on the same or similar hardware, it does not mean that one will arrive to the same type of conclusions in another environment with different configurations. This is a general limitation regarding this type of testing, as there are far too many different variables and potential combinations to test to be able to draw generalized conclusions that would fit every use case.

7 Conclusions

This thesis set out to compare Redis, Kafka and Pulsar as data ingestion systems in a real-time stream processing pipeline. By evaluating the architecture and general characteristics of the systems, we conclude that choosing the right platform depends on the use case. Redis is a potential solution for streaming pipelines that only require short-term persistent storage due to its in-memory design, while Kafka and Pulsar are designed to handle large amounts of long-term on disk persistence. While all platforms generally have great language and container support, Kafka is seen as the most mature technology in terms of integration possibilities in the overall stream processing pipeline. Redis Streams is the least mature solution and has generally not received much industry attention. These factors are crucial in choosing the right platform for intended use case.

Additionally, the paper set out to compare all platforms based on throughput and end-to-end latency performance using 1 kB, 65 kB and 600 kB message sizes. The results show that Kafka is the most consistent platform regarding overall throughput levels. Pulsar performs great at small message sizes but struggles to achieve high throughput with larger messages due to a bottleneck in the testing configuration. The throughput performance of Redis is slightly inconsistent, with low performance for small and very large messages, but performance on par with Kafka for 65 kB messages. In terms of end-to-end latency performance, we conclude that Pulsar is the best performing platform across message sizes at the tested throughput levels. Redis performs very well with lower message sizes but does not scale as well to larger ones. Both Redis and Pulsar manage impressive sub millisecond latencies at low message size and throughput levels. Kafka performs solidly across all the latency tests, slightly worse than both Pulsar and Kafka. In general, all platforms perform well regarding latency.

From an industrial point of view, the different platforms presented and the conclusions drawn in the study will hopefully help companies that are in the process of implementing stream processing pipelines in their product. From a research point of view, the study will work as a base for future research in benchmarking these types of platforms. The study also addresses platforms like Redis Streams and Apache Pulsar, which currently have very little or no presence in current published research.

For future work there is still a need to continue doing performance testing using the same throughput and latency metrics. There are so many different possible combinations in terms of message sizes, throughput levels, hardware, platforms specific configurations etc. that could still be tested. Additionally, there is a need to test the impact that hardware has on overall performance for figuring out the optimal cluster node hardware configurations. For figuring out optimal platform specific configurations, the model developed in [36] for predicting the performance of Kafka could be further expanded on.

References

[1] Confluent, "Kafka commit log," 2020, https://cdn.confluent.io/wp-content/uploads/2016/08/commit_log-copy.png, Last accessed on 2020-02-08.

[2] Amazon AWS, "What is pub/sub messaging?" 2020, https://d1.awsstatic.com/product-marketing/Messaging/sns_img_topic.e024462ec88e79ed63d690a2eed6e050e33fb36f.png, Last accessed on 2020-03-17.

[3] InfoWorld, "How to use redis for real-time stream processing," 2017, https://images.idgesg.net/images/article/2017/08/redis-pubsub-100730561-large.jpg, Last accessed on 2020-03-17.

[4] Devopedia, "Redis streams," 2020, https://devopedia.org/images/article/229/1804.1571239690.png, Last accessed on 2020-03-17.

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

[6] Apache Pulsar, "Architecture overview," 2020, http://pulsar.apache.org/docs/assets/broker-bookie.png, Last accessed on 2020-05-01.

[7] ——, "Messaging concepts," 2020, http://pulsar.apache.org/docs/assets/partitioning.png, Last accessed on 2020-05-01.

[8] ——, "Messaging concepts," 2020, https://pulsar.apache.org/docs/assets/pulsar-subscription-modes.png, Last accessed on 2020-05-01.

[9] MAPR, "Streaming data pipeline to transform, store and explore healthcare dataset with apache kafka api, apache spark, apache drill, json and mapr database," 2020, https://mapr.com/blog/streaming-data-pipeline-transform-store-explore-healthcare-dataset-mapr-db/assets/example-streamline-processing-pipeline.png, Last accessed on 2020-05-01.

[10] Addiva AB, "Addtrack figure," 2020, https://www.addiva.se/wp-content/uploads/2018/03/pastedImage.png, Last accessed on 2020-02-08.

[11] M. Kleppmann, *Making Sense of Stream Processing - The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms.* O'Reilly Media, Inc., 2016.

[12] F. H. Vasiliki Kalavri, *Stream Processing with Apache Flink - Fundamentals, Implementation, and Operation of Streaming Applications.* O'Reilly Media, Inc., 2019.

[13] M. Stonebraker, U. Çetintemel, and S. B. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Record*, vol. 34, pp. 42–47, 2005.

[14] Apache Kafka, "Home page," 2020, https://kafka.apache.org/, Last accessed on 2020-02-04.

[15] Apache Pulsar, "Home page," 2020, https://pulsar.apache.org/, Last accessed on 2020-04-15.

[16] Redis Streams, "Introdution to redis streams," 2020, https://redis.io/topics/streams-intro, Last accessed on 2020-02-04.

[17] Addiva AB, "Addiva software," 2020, https://www.addiva.se/addiva-software/, Last accessed on 2020-02-04.

[18] D. Dedousis, N. Zacheilas, and V. Kalogeraki, "On the fly load balancing to address hot topics in topic-based pub/sub systems," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 76–86.

[19] S. Zeuch, B. Monte, J. Karimov, C. Lutz, M. Renz, J. Traub, S. Breß, T. Rabl, and V. Markl, "Analyzing efficient stream processing on modern hardware," *Proceedings of the VLDB Endowment*, vol. 12, pp. 516–530, 01 2019.

[20] Dave Nielsen, “Popular redis uses for beginners,” Redis Labs, White Paper. [Online]. Available: http://lp.redislabs.com/rs/915-NFD-128/images/WP-RedisLabs-Popular-Redis-Uses-for-Beginners.pdf

[21] Redis, “Redis streams commands,” 2020, https://redis.io/commands#stream, Last accessed on 2020-03-25.

[22] J. Kreps, “Kafka : a distributed messaging system for log processing,” 2011.

[23] M. J. Sax, G. Wang, M. Weidlich, and J.-C. Freytag, “Streams and tables: Two sides of the same coin,” in *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics*, ser. BIRTE ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3242153.3242155

[24] Apache Pulsar, “Pulsar overview,” 2020, https://pulsar.apache.org/docs/en/concepts-overview/, Last accessed on 2020-05-01.

[25] Sharad Murthy, Tony Ng, “Announcing pulsar: Real-time analytics at scale,” eBay, White Paper. [Online]. Available: https://tech.ebayinc.com/engineering/announcing-pulsar-real-time-analytics-at-scale/

[26] Apache Pulsar, “Pulsar functions,” 2020, http://pulsar.apache.org/docs/en/functions-overview/, Last accessed on 2020-05-01.

[27] ——, “Architecture overview,” 2020, https://pulsar.apache.org/docs/en/concepts-architecture-overview/, Last accessed on 2020-05-01.

[28] ——, “Messaging concepts,” 2020, https://pulsar.apache.org/docs/en/concepts-messaging/, Last accessed on 2020-05-01.

[29] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, and S. Khan, “A survey of distributed data stream processing frameworks,” *IEEE Access*, vol. 7, pp. 154 300–154 316, 2019.

[30] M. A. Lopez, A. G. P. Lobato, and O. C. M. B. Duarte, “A performance comparison of open-source stream processing platforms,” in *2016 IEEE Global Communications Conference (GLOBECOM)*, Dec 2016, pp. 1–6.

[31] J. Karimov, T. Rabl, A. Katsifodimos, R. Samarev, H. Heiskanen, and V. Markl, “Benchmarking distributed stream data processing systems,” *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 1507–1518, 2018.

[32] H. Nasiri, S. Nasehi, and M. Goudarzi, “Evaluation of distributed stream processing frameworks for iot applications in smart cities,” *Journal of Big Data*, vol. 6, 12 2019.

[33] P. Le Noac’h, A. Costan, and L. Bougé, “A performance evaluation of apache kafka in support of big data streaming applications,” in *2017 IEEE International Conference on Big Data (Big Data)*, 2017, pp. 4803–4806.

[34] Grid5000, “Home page,” 2020, https://www.grid5000.fr/w/Grid5000:Home, Last accessed on 2020-03-25.

[35] B. Blamey, A. Hellander, and S. Z. Toor, “Apache spark streaming and harmonicio: A performance and architecture comparison,” *CoRR*, vol. abs/1807.07724, 2018.

[36] H. Wu, S. Zhihao, and K. Wolter, “Performance prediction for the apache kafka messaging system,” 08 2019, pp. 154–161.

[37] H. Wu, Z. Shang, and K. Wolter, “Trak: A testing tool for studying the reliability of data delivery in apache kafka,” in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 394–397.

[38] S. Intorruk and T. Numnonda, “A comparative study on performance and resource utilization of real-time distributed messaging systems for big data,” 07 2019, pp. 102–107.

[39] P. Runeson, M. Höst, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering – Guidelines and Examples*, 02 2012.

[40] Addiva AB, "Addtrack," 2020, https://www.addiva.se/addtrack/, Last accessed on 2020-02-08.

[41] Bombardier, "Home page," 2020, https://www.bombardier.com/en/home.html, Last accessed on 2020-02-08.

[42] Redis, "Redis persistence," 2020, https://redis.io/topics/persistence, Last accessed on 2020-03-25.

[43] Apache Kafka, "Apache kafka documentation," 2020, https://kafka.apache.org/documentation/, Last accessed on 2020-05-16.

[44] Apache Pulsar, "Message retention and expiry," 2020, http://pulsar.apache.org/docs/en/cookbooks-retention-expiry/, Last accessed on 2020-05-16.

[45] Redis, "Redis replication," 2020, https://redis.io/topics/replication, Last accessed on 2020-03-25.

[46] ——, "Redis cluster specification," 2020, https://redis.io/topics/cluster-spec, Last accessed on 2020-05-16.

[47] ——, "Redis sentinel documentation," 2020, https://redis.io/topics/sentinel, Last accessed on 2020-05-16.

[48] D. Kjerrumgaard, *Apache Pulsar in Action*. Manning, 2020.

[49] Redis, "Redis clients," 2020, https://redis.io/clients, Last accessed on 2020-03-27.

[50] Docker Hub, "Redis official docker image," 2020, https://hub.docker.com/_/redis/, Last accessed on 2020-03-27.

[51] ——, "Bitnami redis docker image," 2020, https://hub.docker.com/r/bitnami/redis/, Last accessed on 2020-05-15.

[52] Kubernetes, "Home page," 2020, https://kubernetes.io/, Last accessed on 2020-05-15.

[53] Confluence, "Clients - apache kafka," 2020, https://cwiki.apache.org/confluence/display/KAFKA/Clients, Last accessed on 2020-05-15.

[54] Docker Hub, "Kafka bitnami docker image," 2020, https://hub.docker.com/r/bitnami/kafka/, Last accessed on 2020-05-15.

[55] ——, "Kafka wurstmeister docker image," 2020, https://hub.docker.com/r/wurstmeister/kafka, Last accessed on 2020-05-15.

[56] Apache Pulsar, "Pulsar client libraries," 2020, http://pulsar.apache.org/docs/en/client-libraries/, Last accessed on 2020-05-16.

[57] ——, "Pulsar client feature matrix," 2020, https://github.com/apache/pulsar/wiki/Client-Features-Matrix, Last accessed on 2020-05-16.

[58] Apache mail archive, "pulsar-users mailing list archives," 2018, http://mail-archives.apache.org/mod_mbox/pulsar-users/201802.mbox/%3C5a7be0b5.678a6b0a.bb443.e396@mx.google.com%3E, Last accessed on 2020-05-17.

A Setting up the environment

This section will explain the process of setting up each of the environments used in the performance testing. Since the tests were run on a server machine running the Debian 10 linux distribution, the following instructions will reflect that. Any command preceded by a $ indicates that it should be run from the command terminal.

1.1 Docker

Install Docker and Docker-Compose with the following:

```
$ sudo apt install docker.io docker-compose
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

Docker version 18.09.1 and Docker-Compose version 1.21.0 were used.

1.2 Redis

The official Redis image has been used for setting up the Redis server. The following command pulls the image if it has not yet been downloaded locally, and maps the port 6379 on the host to the container, where the Redis server runs by default. The –appendonly directive means that Redis will run with AOF persistence.

```
$ docker run --name redis -p 6379:6379 -d redis redis-server --appendonly yes
```

Confirm that the image is running by typing "docker ps -a". To delete a running container run "docker rm -fv name of container".

1.3 Pulsar

The Pulsar environment is setup in standalone mode using the official Docker image, i.e. one broker, one zookeeper node and one bookkeeper node running in the same container. Start it by running the following:

```
docker run -d --name pulsar -p 6650:6650 -p 8080:8080 \
apachepulsar/pulsar:latest bin/pulsar standalone
```

Create the partitioned topic with two partitions which will be used for testing with the following:

```
$docker exec -d pulsar bash bin/pulsar-admin topics \
create-partitioned-topic persistent://public/default/testing -p 2
```

1.4 Kafka

The Kafka environment is setup using the Docker images by Bitnami. Since a Kafka deployment consists of at least one zookeeper node and one broker, Docker-Compose will be used to easily spin up and tear down an environment with multiple containers. First create a docker-compose.yaml file with the contents seen below. Then run the following command from the same directory that contains the .yaml file.

```
$ docker-compose up -d
```

Create a topic with two partitions called "testing":

```
$ docker exec -d kafka_kafka_1 bash /opt/bitnami/kafka/bin/kafka-topics.sh \
--create --bootstrap-server localhost:9092 \
--replication-factor 1 --partitions 2 --topic testing
```

To tear down the environment, simply type the following:

```
$ docker-compose down
```

The docker-compose.yml file used:

```
version: '2'

services:
  zookeeper:
    image: 'bitnami/zookeeper:latest'
    ports:
      - '2181:2181'
    volumes:
      - 'zookeeper_data:/bitnami'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
  kafka:
    image: 'bitnami/kafka:latest'
    ports:
      - '9092:9092'
      - '29094:29094'
    volumes:
      - 'kafka_data:/bitnami'
    environment:
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - KAFKA_LISTENERS=LISTENER_BOB://kafka:29094,LISTENER_FRED://kafka:9092
      - KAFKA_ADVERTISED_LISTENERS=LISTENER_BOB://kafka:29094,LISTENER_FRED://localhost:9092
      - KAFKA_LISTENER_SECURITY_PROTOCOL_MAP=LISTENER_BOB:PLAINTEXT,LISTENER_FRED:PLAINTEXT
      - KAFKA_INTER_BROKER_LISTENER_NAME=LISTENER_BOB
    depends_on:
      - zookeeper
```

B Running the throughput tests

2.1 Redis

The redis-benchmark tool by default does not support testing the XADD command, which is used to produce events to the stream structure. Because of this, the tests were executed by running a forked version of Redis which includes tests for the XADD command for Redis Streams. This involves downloading the source code and compiling it from source.

```
$ git clone https://github.com/filipecosta90/redis.git
$ cd redis
$ git checkout remotes/origin/benchmark_xadd
$ make install
```

Once compiled, navigate under the /src folder and run the tests from there in the following manner:

```
$ cd /src
$ ./redis-benchmark -t xadd_1 -n {message amount} -h localhost -d {size in byte}
```

2.2 Pulsar

The throughput tests in Pulsar are run using the built-in producer benchmark tester, producing data to the topic that was previously created. This is run within the container by doing the following:

```
$ docker exec -it pulsar /bin/bash
$ cd bin
$ ./pulsar-perf produce --num-messages {messages} --size {size in bytes} \
--rate {msg/sec} persistent://public/default/testing
```

2.3 Kafka

Similar to Pulsar, the tests for Kafka are run within the container using the built-in performance benchmarking tool.

```
$ docker exec -it kafka_kafka_1 /bin/bash
$ cd /opt/bitnami/kafka/bin
$ ./kafka-run-class.sh org.apache.kafka.tools.ProducerPerformance --topic testing \
    --throughput -1 --num-records {messages} --record-size {size in bytes} \
    --producer-props bootstrap.servers=localhost:9092
```

C Running the latency tests

The latency tests have all been written in client libraries for Python, tested with Python version 3.7.3. Install the required client libraries with the specific versions used in the tests:

```
$ pip3 install confluent-kafka==1.4.1 redis==3.5.2 pulsar-client==2.5.1
```

All the latency tests for the different platforms can be run from the same *run.py* script. Put all the code files listed in Appendix D in the same directory using the same filenames. For example, to start two consumers and a single producer for Kafka, with a message size of 1 kB, a message rate of 1000 and a number of 30000 total messages, run the following:

```
$ python3 run.py --platform=kafka --mode=consume --workers=2

$ python3 run.py --platform=kafka --mode=produce --message-count=30000 \
    --bytes=1000 --rate=1000 --workers=1
```

These values can be changed to run the tests on redis and pulsar, as well as with the other message sizes and message rates used.

D Source code for latency tests

```python
import os
from multiprocessing import Pool
import sys
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('--platform', '-p', type=str)
parser.add_argument('--message-count', '-n', type= int)
parser.add_argument('--rate', '-r', type= int)
parser.add_argument('--bytes', '-b', type= int)
parser.add_argument('--mode', '-m', type= str)
parser.add_argument('--workers', '-w', type=int)

args = parser.parse_args()
platform = args.platform
message_count = args.message_count
rate = args.rate
bytes = args.bytes
mode = args.mode
workers = args.workers

def run_producers(process):
    os.system('python3 {} {} {} {}'.format(process, message_count, rate, bytes))

def run_consumers(process):
    os.system('python3 {}'.format(process))

script = f"produce-{platform}.py"
```

```
if(mode == 'produce'):
    processes = ((script),) * workers
elif(mode == 'consume'):
    processes = ((script),) * workers

pool = Pool(processes=workers)

try:
    if(mode == 'produce'):
        pool.map(run_producers, processes)
    if(mode == 'consume'):
        pool.map(run_consumers, processes)

except KeyboardInterrupt:
    pool.terminate()
    pool.join()
```

Listing 1: run.py

```
import redis
import time
import sys
import statistics

HOST = 'localhost'
PORT = 6379

r = redis.Redis(host=HOST, port=PORT)

stream = 'testing'
group = 'consumer-group'

try:
    r.xgroup_create(stream, group, mkstream=True)
except redis.exceptions.ResponseError:
    print("Group already exists")

#holds all the latency calculations
latencies = []

try:
    while True:
        for stream, messages in r.xreadgroup(group, "reader", {stream: '>'}):
            time_received_ns = time.perf_counter_ns()
            time_sent = messages[0][1][b'timestamp'].decode("utf-8")
            time_sent_ns = int(time_sent)

            latency = (time_received_ns - time_sent_ns) / 1000000
            latencies.append(latency)

            msg_id = messages[0][0]

            #make sure that the same msg does not
            #get processed by the other consumers in the group
            r.xack(stream, group, msg_id)

except KeyboardInterrupt:
    print("Consumer shut down.")

finally:
    print(f"{len(latencies)} messages consumed")
    print(f"mean latency: {statistics.mean(latencies)} ms")
    print(f"min latency: {min(latencies)} ms")
    print(f"max latency: {max(latencies)} ms")
```

Listing 2: consume-redis.py

```
import redis
import time
import sys
```

```
HOST = 'localhost'
PORT = 6379

r = redis.Redis(host=HOST, port=PORT)

stream = 'testing'

num_messages = int(sys.argv[1])
msg_rate = int(sys.argv[2])
data_size = int(sys.argv[3])

#Used for throttling the messages/sec based on the given rate
factor = 1 / (msg_rate/1000000000)

#a redis stream value is always a string,
#so have to create a string that is of wanted size in terms of bytes
data_string = 'A' * data_size

#Use the timestamp to calculate latency when message arrives at consumer
data = {"value": data_string, "timestamp": "0" }

start = time.time()
for i in range(num_messages):
    data["timestamp"] = time.perf_counter_ns()
    r.xadd(stream, data)

    stop = time.perf_counter_ns() + factor
    while(True):
        curr = time.perf_counter_ns()
        if (curr >= stop):break


end = time.time()
elapsed = end - start
print("\n\n{} messages sent per second\n\n".format(int(num_messages / elapsed)))
```

Listing 3: produce-redis.py

```
import pulsar, _pulsar
import datetime
import statistics
import time

client = pulsar.Client('pulsar://localhost:6650')

consumer = client.subscribe('persistent://public/default/testing',
                            'consumer-group',
                            consumer_type=_pulsar.ConsumerType.Shared)

#holds all calculated latency values
latencies = []

while True:
    try:
        msg = consumer.receive(timeout_millis=10000)

        # Acknowledge that message has been received
        consumer.acknowledge(msg)

        #calc latency between the sent and received timestamp
        latency = (time.perf_counter_ns() - msg.event_timestamp())/1000000
        latencies.append(latency)

    except:
        print("Closed connection")
        break

print(f"\n{len(latencies)} messages consumed")
print(f"mean latency: {statistics.mean(latencies)} ms")
print(f"min latency: {min(latencies)} ms")
print(f"max latency: {max(latencies)} ms\n")
```

Listing 4: consume-pulsar.py

```
import pulsar
import sys
import time
import datetime

client = pulsar.Client('pulsar://localhost:6650')

producer = client.create_producer('persistent://public/default/testing'
                                    ,block_if_queue_full=True)

num_messages = int(sys.argv[1])
msg_rate = int(sys.argv[2])
data = bytes(int(sys.argv[3]))

factor = 1 / (msg_rate/1000000000)

def send_callback(res, msg):
    return

start = time.time()
for i in range(num_messages):
    producer.send(data, event_timestamp=time.perf_counter_ns())

    #when the throttle delay should be stopped
    stop = time.perf_counter_ns() + factor
    while(True):
        curr = time.perf_counter_ns()
        if (curr >= stop):break

#flush and close producer
producer.flush()
producer.close()

#Basic producer statistics
end = time.time()
elapsed = end - start
print("\n\n{} messages sent per second\n\n".format(int(num_messages / elapsed)))
```

Listing 5: produce-pulsar.py

```
from confluent_kafka import Consumer, TopicPartition
import socket
import time
import statistics
import datetime
import statistics

conf = {'bootstrap.servers': "localhost:9092"}

c = Consumer({
    'bootstrap.servers': conf['bootstrap.servers'],
    'group.id': 'cg',
    'auto.offset.reset': 'earliest',
})


def print_assignment(consumer, partitions):
    print('Assignment:', partitions)

c.subscribe(['testing'], on_assign=print_assignment)
#c.assign([TopicPartition('testing', 0)])

#holds all the latency measurements
latencies = []

try:
    while True:
        msg = c.poll(1.0)
        if msg is None:
            continue

        elif msg.error():
            print('error: {}'.format(msg.error()))

        else:
            # Check for message
            latency =  (datetime.datetime.now().timestamp() * 1000) - msg.timestamp()[1]
            latencies.append(latency)


except KeyboardInterrupt:
    pass
finally:
    c.close()
    print(f"{len(latencies)} messages consumed")
    print(f"mean latency: {statistics.mean(latencies)} ms")
    print(f"min latency: {min(latencies)} ms")
    print(f"max latency: {max(latencies)} ms")
```

Listing 6: consume-kafka.py

```
from confluent_kafka import Producer
import sys
import time
from functools import wraps


if __name__ == '__main__':

    broker = "localhost:9092"
    topic = "testing"

    conf = {'bootstrap.servers': broker}

    # Create Producer instance
    p = Producer(**conf)

    num_messages = int(sys.argv[1])
    msg_rate = int(sys.argv[2])
    data = bytes(int(sys.argv[3]))

    #used for throttling based on the wanted rate
    factor = 1 / (msg_rate/1000000000)

    def delivery_callback(err, msg):
        if err:
            sys.stderr.write('%% Message failed delivery: %s\n' % err)

    def produce(p, topic, data, cb):
        p.produce(topic, data, callback=cb)


    i = 0
    start = time.time()
    while(i < num_messages):
        try:
            produce(p, topic, data, delivery_callback)
            stop = time.perf_counter_ns() + factor

            i+=1
            while(True):
                curr = time.perf_counter_ns()
                if (curr >= stop):break

        except BufferError:
            sys.stderr.write('Local producer queue is full!')
        p.poll(0)

    # Wait until all messages have been delivered
    p.flush()

    end = time.time()
    elapsed = end - start
    print("{} messages sent per second".format(int(num_messages / elapsed)))
```

Listing 7: produce-kafka.py