Thialfi & Kafka: A Comparison between Messaging Systems

Malte Granderath
Technical University Munich
Munich, Germany
malte.granderath@tum.de

Adrian Koegl
Technical University Munich
Munich, Germany
ga84vuw@mytum.de

ABSTRACT

Many applications have a need for real-time updates, whether it is for processing data or for displaying updated information to clients. The systems that enable this kind of property often develop with different properties and architectures, although their general objective can be seen as very similar.

This paper is a comparison between two of these systems, Thialfi and Kafka. We detail the objectives, architecture and properties of these systems and provide a comparison between them. Hereby we focus on how existing problems are solved by these systems and how their selected communication model enables them to succeed.

KEYWORDS

distributed systems, messaging, thialfi, kafka

1 INTRODUCTION

In the last years, the use of the internet has considerably changed. Many of the applications that were previously running as native applications are now often built as web applications. With this change, many users expect high interactivity from web applications and that requires a set of distributed systems that enable this goal. These distributed systems now have to handle high volumes of different entities, potentially spread all over the world, and whose behavior greatly varies. Traditional systems that follow point-to-point and synchronous communication can not support dynamic applications, so there is a demand for more flexible communication models and systems.

The *publish-subscribe* communication model has seen increased usage in systems that are aimed at allowing the loose coupling of communication for real-time systems. It is a simple model where interested clients can subscribe to a pipe and receive the information that publishers push into the pipe. Due to this model's flexible nature, there have been many different variations developed for different use cases. In this paper, we compare two different systems built around the *publish-subscribe* model but are using this in very different manners to achieve their designed goal.

The rest of the paper is structured as follows. In the Section 2 we provide an introduction to the publish-subscribe pattern. In Section 3 we provide an in-depth look at Thialfi; similarly, we take a look at Kafka in Section 4. We conclude the paper with a comparison between the two systems in Section 5.

2 PUBLISH-SUBSCRIBE

The publish-subscribe pattern allows *subscribers* to express their interest in specific events or a group of events and following this, be notified of any new event that matches their interests. These events are generated by *publishers*. The specific events or groups of events are called *topics*. Publishers publish on specific topics as a reaction to some other event that they receive. For example, user events on a web application. The subscribers have explicitly subscribe to specific topics and then, usually, get the new events pushed to them instead of continuously checking for new events.

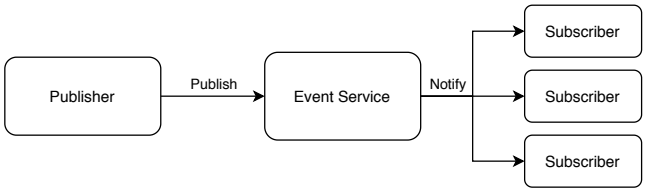

Figure 1: Publish-Subscribe Model

The simplified system model for the publish-subscribe pattern is dependent on a service that is handling the storage and management for subscriptions, as well as the efficient delivery of the events. This is called the Event Service in Figure 1.

The event service provides the decoupling in three dimensions between the publisher and subscriber:

- **Space**: The interacting components do not need to know each other
- **Time**: The interacting components do not actively have to participate in the exchange of events. For example, the subscriber can be offline at the time that the publisher publishes a new event.
- **Synchronization**: All of the operations take place in an asynchronous manner. Publishers do not get blocked by waiting for subscribers and subscribers receive the new events asynchronously.

This provides many benefits to real-time systems that try to achieve high throughput while maintaining low latency. Additionally, the flexibility of the publish-subscribe model allows for the use with many different target applications. This will become more apparent in the next sections where we show the use of the publish-subscribe model for Thialfi and Kafka [4].

3 THIALFI

Thialfi is being developed and used internally at Google. Currently, the system is closed source. Google describes it as a "Notification Service," but more specifically, it is an *object update signalling service*. The system was built around Google's need for real-time features in many of their consumer-facing applications, such as Chrome bookmarks, where real-time data is important but not part of the primary aspects of the application. Previously many applications succeeded in doing that by continuously polling for updates to the application backends, which may cause delays and higher server load [3].

Google, having many different client applications and millions of users, was focused on developing a system that none of the applications would need to be rebuilt entirely for and could be used with a wide variety of different styles of applications. Additionally, Google needs a system that can scale well with its high number of users while still maintaining low latency. This has influenced the majority of the design decisions that were made and will be described in this section.

In the following sections, Thialfi will be presented by first giving an overview of the general solution, then at the architecture. Finally the properties will be analyzed and evaluated.

3.1 Overview

Thialfi uses the publish-subscribe model with the variation that clients subscribe for specific object updates. The object updates are the events that flow through the system, but they are published as pairs of object ID to the version number of that object. The version number is how Thialfi and the client detect that there is a new version available. Thialfi does not carry the object value but is built around notifying that an object has been updated without giving specific details about the update. In this context *client* refers to applications running on consumer devices and *backend* to the application backend. In the Thialfi specification, the subscribe operation from publish-subscribe is referred to as *register*.

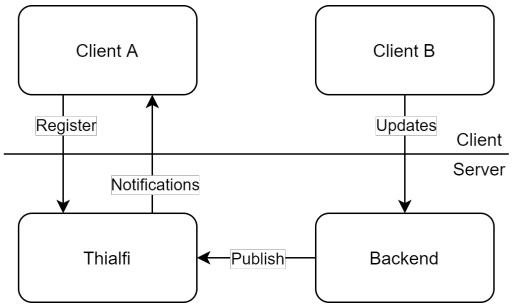

Figure 2: Thialfi Communication

Thialfi is an additional component to the usual communication pattern in these applications. Applications still need to fetch the data directly from the backend when it has to be displayed but then register their interest based on the IDs of the objects that they fetched. In Figure 2, the generalized communication pattern of clients, the backend and Thialfi is visualized. A more specific process description is the following:

(1) Application fetches objects from backend.
(2) Application registers for updates of specific objects using their IDs.
(3) Once the backend receives a request to update an object it publishes an object update to Thialfi.
(4) Thialfi forwards the update notification to all clients registered for that object ID.
(5) If the application receives a notification it will re-fetch the object from the backend.

Thialfi can be seen as the event service component of the publish-subscribe model as referenced in Section 2.

Thialfi utilizes two libraries to simplify registering and publishing object updates. They are called the *client* and *publisher* library. In Figure 2, the client library is used in client A and the publisher library as part of the backend. A simplified API of these two libraries, that still captures the main idea, is that the clients subscribe using `register(objectId)` to register for updates of a specific object ID and the backends can publish updates using `publish(objectId,versionNumber)` [3].

3.2 Architecture

Thialfi has to support millions of different subscriptions, so the architecture was designed for precisely this use. All of the components can be scaled horizontally according to the needs in that timeframe [3].

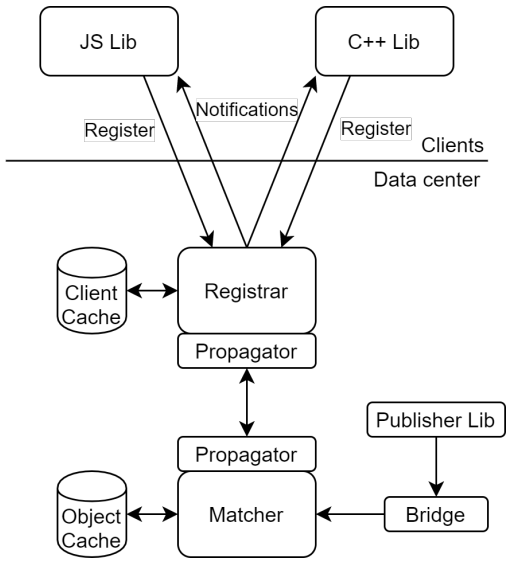

Figure 3: Thialfi Architecture.

Figure 3 shows the different components of Thialfi and they will be described in more detail below. The **Bridge** component consumes the stream of object updates created from the Publisher library's usage and assembles them into batches for the delivery to the Matchers. **Matchers** consume the notifications from the Bridge and forward them to the relevant Registrars. The **Registrar** is the point of contact for the actual client, so they handle client registrations, track the clients and reliably deliver notifications to the client. For scalability, the Registrars are partitioned over the client IDs and the Matchers are partitioned over the object IDs.

The **Client Cache** holds two sets for each client:

(1) Registrations: objects the client has subscribed for
(2) Pending notifications: notifications that have not been acknowledged by the client

The **Object cache** holds the latest version number for each object provided by the back end and a secondary copy of the registered clients for each object.

Another small but very critical component of Thialfi is the **Propagators**. The Registrar and Matcher never directly communicate

with each other but instead push the information they want to send into a pending operation set. The Propagators then asynchronously forward this information to each other. This has the benefit that there is no blocking on the Matcher and Registrar when waiting for a delivery to succeed.

3.3 Reliable Delivery

Reliable delivery has been mentioned as part of the Registrar's responsibility, but reliable delivery is a property that has to be ensured by the entire service. In the context of Thialfi, reliable delivery is defined as:

> If a well-behaved client registers for an object X, Thialfi ensures that the client will always **eventually** learn of the latest version of X

Figure 4: Reliably Delivery Property [3]

This means that there is no upper bound of time for a notification to be delivered. Instead, Thialfi is focused on delivering the notification at some point in time after the backend has published it. To achieve this property, Thialfi ensures that the state in one component is eventually propagated to all relevant other components. The two paths of propagation are shown below.

The **Registration state** is propagated from the client to the Registrar and then the Matcher.

- **Client ↔ Registrar**: All messages from the client contain a digest of its registration state. The client, additionally, sends continuous heartbeats. The Registrar can then compare this to its state and start a new message sequence to synchronize the registration state of the client.
- **Registrar → Matcher**: When the Registrar changes its registration state, it is also marked as a pending operation. The Registrar propagator will then make sure that the update is propagated to the Matcher.

The **Notificate state** is propagated from the Publisher through the Bridge, Matcher, Registrar and finally the client.

- **Bridge → Matcher**: Notifications are only removed from the update feed by the Bridge once they have been successfully received at the Matcher.
- **Matcher → Registrar**: Once a notification has been received by the Matcher it is marked as a pending operation. The Matcher propagator will then forward the notification to the relevant Registrars.
- **Registrar → Client**: The Registrar resends the notification until it has been acknowledged by the client or there is a superseding notification for that object with a higher version number.

3.4 Fault Tolerance

There are two methods that allow for high fault tolerance with Thialfi. One of the approaches is based upon the complete rebuilding of all state and the other by using *BigTable* for persistent storage.

In the naive approach, all of the components are restarted when one of the components crashes. The discrepancy between the Registrar's registration state and the client's registration state will then cause synchronization of the registration and propagation to the Matcher. If there is no version number available for an object at the Matcher, it will send a notification with the version number specified as unknown and this will cause the client to re-fetch the object from the backend. Therefore, correctness is still guaranteed using this approach.

The naive approach works well on a smaller scale, but with millions of clients have to resynchronize their registrations a lot of unnecessary load is put on the registrars. As a solution to this problem, a database can be used behind the cache with *blind writes*. Blind writes refer to writing to the database without reading. Google uses *BigTable* as their database of choice. Once a component restarts it can read the cache from the database and continue. In case there are any problems with the database Thialfi can fall-back on the naive method to recovery [3].

3.5 Evaluation

Google developed Thialfi as a solution to many of the problems they were facing regarding their applications. As many of their applications had developed their own approach to including real-time updates, one of the higher priority focuses is the ease of integration. As Thialfi is simply an add-on to the existing application communication pattern and does not require restructuring applications for its use, it succeeds in this aspect.

With many of its applications having millions of users, Google needs Thialfi to be readily scalable. From its internal use in Google, Thialfi has been shown to scale to millions of users and resource consumption following a linear correlation with the number of users. Additionally, the notification latencies are stable concerning the notification rate [2, 3].

As Thialfi is meant to deliver real-time notifications, performance is of high importance. From their own measurements, about 88% of notifications are delivered in less than 1 second. This could still be improved by reducing the amount of batching of the notifications, but since Thialfi replaces many systems previously using polling, this is an acceptable value.

The robustness of Thialfi against failures is again of high importance. Thialfi is highly fault-tolerant even with regards to whole data-center failures. This is achieved by having the state distributed across both the server and the clients. This means that the clients can drive recovery in case of failures [2].

4 KAFKA

Modern websites generate a lot of user-driven data, which is potentially required as an input for many services in real-time in order to perform optimally. The requirement of real-time data access can be met by distributed streaming platforms through providing data pipelines. These pipelines generally enable to process high volumes of data as soon as it is made available [10]. Nevertheless, different approaches to realizing distributed streaming platforms have varying performance goals [9]. Therefore, these solutions aren't necessarily streamlined for any purpose.

In 2009 Goodhope et al. researched for such a distributed streaming platform at LinkedIn because their point-to-point pipelines didn't suffice the real-time availability requirement. So they tried

to find a fitting solution for their growing site complexity and increasing volume of data. As it consisted mainly of user activity data, one special demand for their system was the capability to provide high throughput. After all tested products turned out to have issues with their throughput capacity, because they rather focused on low-latency, they decided to build their own messaging service called Kafka [5].

In the following sections, Kafka will be presented by first providing an overview of the whole system briefly explaining all the different components. As Kafka differs from other systems in its high throughput capability, the performance considerations and enhancements are discussed in more detail afterward.

4.1 Overview

Kafka is a distributed streaming platform that is able to transmit streams of messages in a fault-tolerant and scalable manner. These streams are possibly persisted and available in real-time and are reliably exchanged between applications making use of the publish-subscribe pattern. These properties allow many different application possibilities. For example, it can be used as a messaging system, storage system or stream processor.

4.2 Topics

The foundation of realizing the publish-subscribe pattern are *topics* with two possible operations: append and fetch.

Topics can be generally understood as a category of messages which structure is similar to a message queue. Although an essential difference to the primary use of queues is that elements of a topic aren't removed on consumption. Every element in a topic of Kafka is a byte array or respectively a record consisting of a key, value, and optionally timestamp and header. The value can be any kind of data which needs to be processed. The key can be provided by the producer in order to choose where a record shall be located; more details are discussed in Section 4.5. When a new record is appended to a topic, an offset is assigned to it, where the offset is an integer incrementally increasing for every entry.

4.3 Fundamental Architecture

The essential components of Kafka, interacting in a publish-subscribe pattern and using topics, are the *producers*, *brokers* and *consumers*. The core infrastructure consists of one or more brokers, namely the *cluster*. It provides an API through which producers can publish messages as a record and consumers can fetch them.

The producer is an application which generates data and may publish it as a record to a selected topic on a broker. These records are then processed by one or many consumers. This is done by consumers first subscribing to the topics it wants to receive data from and then continuously fetching records from them. In many publish-subscribe systems, a single consumer process subscribes to topics and processes all data contained in these topics [4]. Kafka tries to handle user activity data in real-time, which may be of very high volume speaking of over "300 million members' activities every day"[11]. Therefore they require more processing power for one topic than the standard publish-subscribe model would be able to provide.

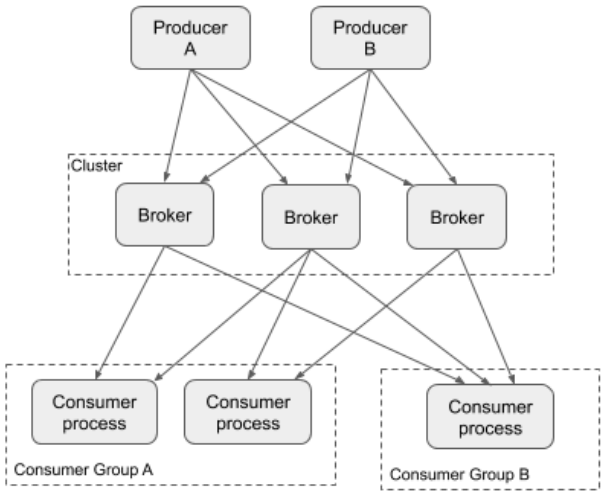

Figure 5: Kafka Architecture

For this reason, Kafka combines the benefits of publish-subscribe and *queuing* models. In a queue, every message is delivered to only one consumer. This has the benefit of distributing processing power over several instances. On the other hand, the publish-subscribe pattern provides a message to all subscribed consumers and allowing various operations on the same data by different processes. Combining workload distribution and different operations, Kafka introduces *Consumer groups*.

Consumer groups contain one or more consumer processes, where the whole group subscribes on topics. Records of a subscribed topic are then distributed in between a group in order to balance load. This is illustrated in Figure 5, where Consumer Group B has only one process, because the topic it subscribed on and the operation performed on the data might not require too much processing power, while Consumer Group A consists of two processes for balancing expensive tasks.

4.4 Partitioning of Topics

To realize the distribution of topics across all brokers they are partitioned and every consumer process within a group is responsible for a set of partitions and no other process in the same group is allowed to consume from it. This allows a partition to be the unit of parallelism and therefore consumer processes belonging to a group can work simultaneously on the same topic.

These partitions are as well distributed across all brokers, where every broker is responsible for zero or more partitions for each topic. This leads to the further advantage of one topic being allowed to contain more data than one machine would be able to. The assignment of partition sets on broker and consumer side are dynamically handled by *Zookeeper*, which is "a service for coordinating processes of distributed applications" [6]. Every time a consumer or broker disappears or appears a re-assignment of partitions is triggered, handled by Zookeeper.

Figure 6 shows an example partitioning of a topic. Here the topic is partitioned into four different partitions. In addition to the shown partition responsibilities of Broker 0 and Broker 1, there also might

be a Broker 2 not in charge of any partition. This example also shows every partition has their own per message incrementally increasing offset number.

Partitions are as well replicated across a predefined amount of brokers for the objective of providing fault-tolerance. Every partition has one leader and follower. The leader processes all read and write requests of producers and consumers, while the followers just replicate the topic. If the leader is not available anymore one of the follower will simply take its place.

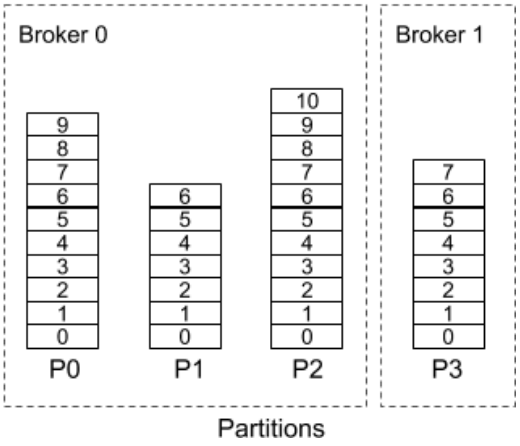

Figure 6: Example topic partitioning

4.5 Key-based Partitioning

A use-case of Kafka is the analysis of user data. For this purpose, the data of a specific user must be available at one consumer in order to analyze the events in the context of all events of a user. Otherwise there would be a need of aggregating all data on consumer side and then picking user corresponding records, what would make a distributed messaging service obsolete. We therefore require the possibility to somehow ensure, that all data belonging to a user will be contained in only one partition.

Within a partition there is obviously an absolute order as new incoming records are simply appended and an incrementally increasing offset is assigned. Providing an absolute order between different partitions isn't possible, but it is feasible to assign messages to partitions according to some attribute. This can be done by *key-based partitioning*.

Key-based partitioning is the process of hashing a message over a key to receive the according partition number to send a record to. This computation is done at the producer to figure out the target partition and therefore also which broker to send the record to. The key may be provided by the application, as in example the user id to the producer through the API. If the producer doesn't submit a key as parameter, the destined partition is simply selected at random.

4.6 Consumer Position

In the previous sections it was mentioned that consumers "fetch" records from the broker. This way of consumers pulling messages of brokers is a very Kafka-specific decision. Normally a publish-subscribe system's interaction includes brokers pushing messages to the consumers [4]. This leads to the fact that in common publish-subscribe implementations the consumer doesn't keep state and just receives data to process.

Implementing a stateful consumer enables several advantages. The first obvious one is, that the consumer can choose which records to fetch. It can simply jump back and forth as necessary since it has the state of its current offset position. As the consumer keeps state of which messages it has processed, the broker doesn't have to keep several offset positions per consumer for every partition. However the cluster has knowledge about the last processed offset for every consumer. So whenever the consumer has processed a message it can send an acknowledgment to the cluster. Consequently messages will be processed starting from the last acknowledged offset if a consumer crashes or another consumer takes responsibility over a certain partition. Furthermore the performance is improved because the consumer doesn't get potentially overwhelmed by too high volume of pushed data, but decides depending on the current capacity when to pull new records.

4.7 Performance Considerations

In comparison to other distributed streaming platform solutions "the primary design criteria for [Kafka] is maximizing the end-to-end throughput while providing low latency and reasonable semantics to the system users"[5]. The goals of low-latency and high throughput are a trade-off [8], which is why it was required to focus on one of them. This leads to the result, that while designing Kafka latency was given up for higher bandwidth capacity usage. Nevertheless the developers tried to keep the latency as low as possible.

Additionally the system has to be built such that the performance is independent of the consumers behavior. Slow fetching consumers, as in example third party applications, may increase the amount of unconsumed records. Therefore it is required to keep both properties constant to the volume of unconsumed data. In order to meet these goals and minimize the trade-off as much as possible a few techniques are implemented. Two of them with the highest impact on performance [5] are explained hereafter.

The first technique is *Batching*. Batching can be made use of before sending messages or writing them to disk. At these steps a certain amount of messages are collected and the operation is performed on all of them at once. To allow this performance improvement the producer and consumer APIs provide an asynchronous send or respectively fetch method. The particular operation is performed when a configured amount of messages has been gathered or if a timeout elapsed. In a benchmark by LinkedIn the throughput could be improved by a factor of 3.2 with a buffer size of 200 messages [7]. However the key bottleneck which needs to be optimized in many use-cases is the throughput between different data centers. Therefore the technique applied here has the greatest impact on performance. Namely Kafka is *shrinking data* in order to reduce the data size transmitted among data center facilities and therefore using the bandwidth more efficiently. "Efficient compression requires compressing multiple messages together rather than compressing each message individually" [1]. Kafka implements different compression algorithms to support shrinking batches of messages. This lot of records is persisted in its compressed state on

	Thialfi	Kafka
Performance optimization	asynchronous message processing	
Messaging pattern	publish-subscribe	
Subscription mode	object-based subscription	category-based subscription
Fundamental functionality	object update notification	data stream processing
Fault-tolerance	database & client state restoration	topic replication across brokers
Integration level	extension communication layer	base communication layer
Availability	Proprietary	Open source

Table 1: Thialfi vs. Kafka comparison

the broker and is consequently only decompressed at the consumer.

5 COMPARISON

As we have looked at the messaging services Thialfi and Kafka separately, we will now have a closer look into the similarities and differences.

Both systems are solving very different problems. Nonetheless, as already introduced, these newer issues mainly arise through the requirement of increasingly high interactivity as well as high volumes of data. To meet them the performance decisions made are quite similar, while still consisting of very heterogeneous resulting structures. In Table 1 the differences are abstractly presented such that the main variations, which emerged through a different problem statement, become apparent. Nevertheless, most of these comparisons call for further discussions.

Neither Thialfi nor Kafka have realised the common structure of the publish-subscribe pattern as we have introduced it in section 2. Thialfi needs to adapt the topic model to their object-based subscription, while Kafka introduces the concept of consumer groups. This means that consumers in Kafka do not have to have knowledge of specific published objects, whereas in Thialfi the object ids must be known and the indirect publisher is usually also the subscriber. This basic difference also illustrates the main purpose of both systems: Kafka tries to improve data stream processing focusing on categories of data and Thialfi notifies subscribers on object updates to enable cooperative object modification.

Something which we didn't mention during the introduction of Kafka, but becomes relevant when comparing both systems, is the way of practically implementing the two systems. Kafka requires to built all functionality around it and can be seen as the base communication layer. Thialfi, on the other hand, is just an extension to an existing system. As a result there is not a lot of adaptations needed to integrate Thialfi.

6 CONCLUSION

The two presented systems depict a very realistic trend for currently and further developing messaging services facing the new requirements of the internet. They provide the ability of coping with the tremendously increasing volume of data, as well as supporting real-time applications.

In order to show the general approach of decoupled messaging models we have initially introduced the publish-subscribe pattern. Starting with the proprietary solution Thialfi, used as object update signalling service at Google, its structure and all components with focus on the implementation of reliable delivery was explained. After that the open sourced distributed streaming platform Kafka was presented with its architecture, specific design decisions and performance considerations. As a conclusion we outlined the differences in both approaches, especially in implementing the publish-subscribe pattern.

In summary we have seen flexible ways of enabling communication with respect to the new circumstances and ingredients of modern websites. Although these messaging services were built on problems in different domains, having an overview and understanding of the two architectures provides an idea of how to adapt to the changing requirements of the internet.

REFERENCES

[1] 2017. Documentation. http://kafka.apache.org/documentation/

[2] Atul Adya. 2013. *Lessons from an Internet-Scale Notification System.* http://2013.ladisworkshop.org/slides/keynote_adya.pdf

[3] Atul Adya, Gregory Cooper, Daniel Myers, and Michael Piatek. 2011. Thialfi: a client notification service for internet-scale applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 129–142.

[4] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The Many Faces of Publish/Subscribe. *ACM Comput. Surv.* 35, 2 (June 2003), 114–131. https://doi.org/10.1145/857076.857078

[5] Ken Goodhope, Joel Koshy, Jay Kreps, Neha Narkhede, Richard Park, Jun Rao, and Victor Yang Ye. 2012. Building LinkedIn's Real-time Activity Data Pipeline. *IEEE Data Eng. Bull.* 35, 2 (2012), 33–45.

[6] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free Coordination for Internet-scale Systems.. In *USENIX annual technical conference*, Vol. 8.

[7] Jay Kreps. 2014. Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines). https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

[8] David A. Patterson. 2004. Latency Lags Bandwith. *Commun. ACM* 47, 10 (Oct. 2004), 71–75. https://doi.org/10.1145/1022594.1022596

[9] S. Qian, G. Wu, J. Huang, and T. Das. 2016. Benchmarking modern distributed streaming platforms. In *2016 IEEE International Conference on Industrial Technology (ICIT)*. 592–598.

[10] Jonathan Samosir, Maria Indrawan-Santiago, and Pari Delir Haghighi. 2016. An Evaluation of Data Stream Processing Systems for Data Driven Applications. *Procedia Computer Science* 80 (2016), 439 – 449. https://doi.org/10.1016/j.procs.2016.05.322 International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA.

[11] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a Replicated Logging System with Apache Kafka. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1654–1655. https://doi.org/10.14778/2824032.2824063