POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Kabis: a platform for event-based communication with configurable trade-off between trust guarantees and performance

Laurea Magistrale in Computer Science Engineering - Ingegneria Informatica

Author: **Matteo Secco**

Advisor: **Prof. Alessandro Margara**

Academic year: **2022-2023**

---

1. Introduction

Event-based interaction has been advocated as the right communication pattern in modern microservices architectures [7]. In event-based interaction, processes execution depends on events that happened into the system.
To propagate state changes across services, event sourcing [3] has gained popularity as a data storage model that logs the whole history of events that led to the current system state. These logs become the persistent memory of the evolution of the system, and the state of each system component at a given point in time can be always derived from the logs, by re-executing the events it stores. As a result, event sourcing has become a common pattern in service-based cloud applications, supported by cloud providers such as Microsoft Azure [5], Amazon AWS [1], and Google Cloud [4]. Event-sourcing architectures have the benefit of allowing client processes to only read from the log when they are compu-

tationally able of handling a new event.
Technologies addressing such failures exist, but their use is limited due to their high performance cost. Another desired property is non-repudiation, which is the impossibility for the process that notified an event to dispute its authorship and any responsibility derived from the notification itself.
In this scenario Kabis was developed, an event-sourcing system which can work in presence of malicious processes. Kabis can be configured to provide two different levels of guarantees at runtime, and allows parties to validate the correctness of the execution asynchronously, thus achieving the non-repudiation property..

2. System model

Kabis is designed to mimic the architecture of Apache Kafka, while optionally exploiting BFT-SMaRt to provide stronger functional guarantees in exchange of a performance overhead, offering to clients the possibility to modify the desired behavior at run-time.
The level of guarantees obtained for an event consumption depends on the combined configuration of both the producer and the consumer of the event.

	Unvalidable	**Validable**
Don't validate	Basic	Basic
Validate	Not delivered	Extended

Table 1: Class of guarantees by producer's and consumer's behavior

Basic guarantees are the very same of Kafka:

- FIFO order on partition level.
- Total order on a partition level.
- Crash fault tolerance.

Extended guarantees include all of the basic guarantees, plus the following:

- Byzantine fault tolerance.
- Non-repudiation.

3. Kabis event service

Kabis inherits Kafka's architecture of partitioned, persisted, replicated topics. Client processes are divided into two kinds: `KabisConsumer` that consume events from the topics, and `KabisProducer` to publish events on them. `KabisConsumers` and `KabisProducers` are strongly decoupled. Kabis is a multi-consumer and multi-producer system.

3.1. KabisProducer API

`KabisProducer` API has been intentionally designed as a subset of the Kafka `Producer` API. Kafka `Producer` API can be clustered as follows:

1. Transaction management, to make reads and writes transactional.
2. Metadata gathering, to retrieve information about the system.
3. Sending methods, to publish events.
4. Disposing methods, for termination.

KabisProducer API handles all the transaction management internally, while the other clusters could in principle be implemented. The correspondence between Kafka and Kabis implemented methods is shown in table 2.

	Kafka	**Kabis**
1	initTransactions() beginTransactions() sendOffsetsToTransaction(∗)[1] commitTransaction() abortTransaction()	
2	partitionsFor(String topic) metrics()	
3	send(ProducerRecord) send(ProducerRecord,Callback) flush()	push(KabisRecord) flush()
4	close() close(Duration)	close() close(Duration)

[1] For space and readability, not implemented methods' arguments have been replaced by the ∗ symbol. This also allowed to collapse overloaded methods.

Table 2: Producer API of Kafka and Kabis

3.2. KabisConsumer API

As for the `KabisProducer` API, the `KabisConsumer` API consists of a subset of Kafka `Consumer` API:

1. Metadata gathering.
2. Topic navigation, to move the reading position on the topic.
3. Execution suspension, for pausing the `Consumer`.
4. Offset commit, crucial for error recovery.
5. Rebalance, for exceptional usage.
6. Subscription management, to subscribe to topics.
7. Poll, to fetch new data.

8. Disposing methods.

KabisConsumer API only expose methods from clusters 6, 7 and 8. Clusters 1, 3, 5 could easily be added to the system, while the other clusters have been left out by design.

	Kafka	**Kabis**
1	assignment() subscription() committed(∗)[1] metrics() partitionsFor(∗)[1] listTopics(∗)[1] paused() offsetsForTimes(∗)[1] beginningOffsets(∗)[1] endOffsets(∗)[1] groupMetadata()	
2	seek(∗)[1] seekToBeginning(∗)[1] seekToEnd(∗)[1] position(∗)[1]	
3	pause(∗)[1] resume(∗)[1] wakeup()	
4	commitSync(∗)[1] commitAsync(∗)[1]	
5	enforceRebalance()	
6	subscribe(Collection<String>) subscribe(∗)[1] assign(Collection) unsubscribe()	subscribe(Collection<String>) unsubscribe()
7	poll(Duration)	pull(Duration)
8	close() close(Duration)	close() close(Duration)

[1] For space and readability, not implemented methods' arguments have been replaced by the ∗ symbol. This also allowed to collapse overloaded methods.

Table 3: Consumer API of Kafka and Kabis

4. System design and implementation

Kabis is designed as two independent communication channels. The *storage channel* is used for data communication, while the *validation channel* can be used for message ordering. It contains *Secure identifiers (SID)*, obtained through digital signature. This allows to keep the bandwidth on the *validation channel* constrained to a constant value.

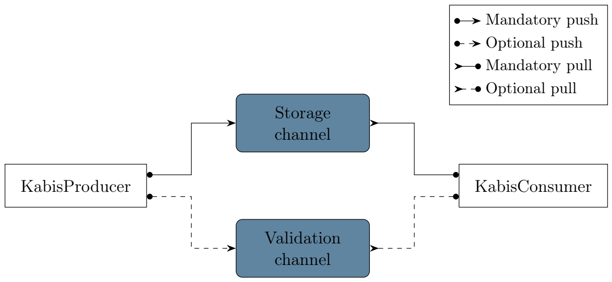

Figure 1: Kabis components.

4.1. Storage implementation and deployment

To be able to resist to F_C crash failures or F_B byzantine failures, the *storage channel* is developed as F_B+1 subsystems called *Kafka replicas*, each consisting of F_C+1 Zookeeper instances and F_C+1 Kafka brokers. Events are published wrapped in a `MessageWrapper<V>` object, which enriches the original event representation with an identifier of the event publisher.

To ensure maximum fairness, each *Kafka replica* should be evenly distributed among the parties.

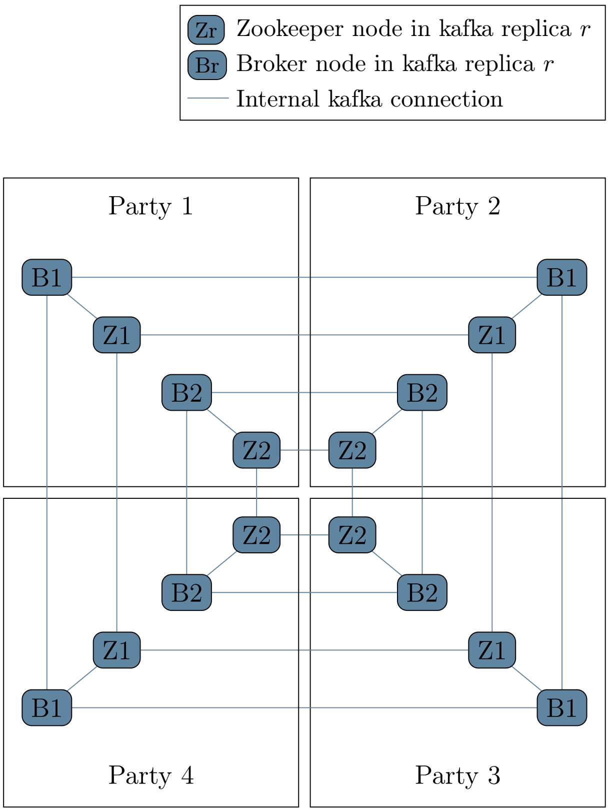

Figure 2: *Storage channel* distributed among 4 parties, tolerant to at most 1 byzantine failure or 3 crash failures.

4.2. Validation implementation and deployment

The *validation channel* is a custom implementation of the BFT-SMaRt system deployed with $3F_B+1$ *service replicas* to tolerate F_B byzantine failures.

Data is sent through this channel as a `SecureIdentifier` (SID) object, encoding the event `topic` and `partition`, an identifier of the event producer, and the digital signature of the

`key`,`value`,`topic` and `partition` of the corresponding `MessageWrapper` sent through the *storage channel*.
Each party using the system will own exactly one *service replica* and a *service proxy* for each client process the party owns.

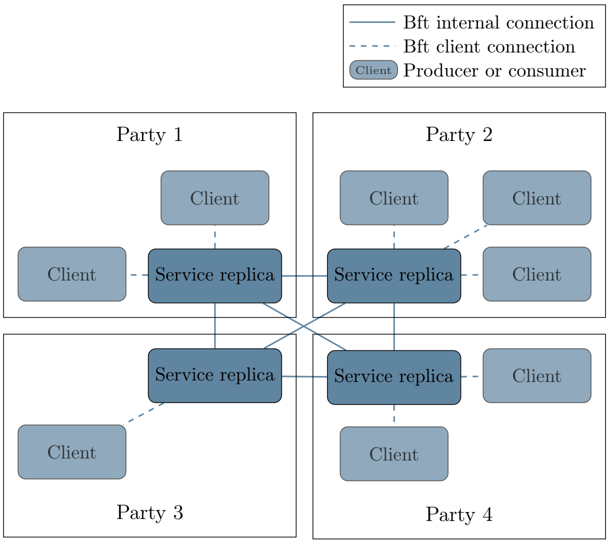

Figure 3: *Validation* component distributed among 4 parties, tolerant to at most 1 byzantine failure or 3 crash failures.

5. Network protocols

The two main operations of Kabis APIs are the `void push(KabisRecord)` used to publish events, and `Iterable<KabisRecord> pull(Duration)` to receive event notifications.

5.1. Push protocol

This section will focus on the case of pushing a *validable event*. *Unvalidable events* are also introduced, but not discussed in detail since the algorithm it's almost identical to a standard Kafka usage.
When receiving the record, the `KabisProducer` first pushes computes the partition and the signature and creates the `MessageWrapper` and the `SID` , that are sent respectively to the *storage channel* and to the *validation channel* in parallel.

Unvalidable events are published by sending the `MessageWrapper` only through the first *Kafka replica*.

5.2. Pull protocol

`KabisConsumers` consume events with the `Iterable <KabisRecord> pull(Duration)` primitive. This involves two sub-processes:

KafkaPollingThread is a cache of Kafka events, indexed for *Kafka replica* and by pairs of `TopicPartitions` and `KabisProducers`, which automatically and periodically pulls each *Kafka replica*. For each replica the cache tracks the (presumably) failed `KabisProducers`.
By defining `isFull(replica)` as a predicate which is true for *Kafka replica* r if, and only if, for each pair of `KabisProducer` p and `TopicPartition` t, either p never sent any event on t or the queue identified by the tuple $< r, t, p >$ contains at least one event. Then the pseudo-code for an iteration of the `KafkaPollingThread` is:

Algorithm 1 `KafkaPollingThread` main loop

```
$K \leftarrow$ set of *kafka replicas*
for all $k \in K$ do
    if $!isFull(k)$ then
        $C \leftarrow getKafkaConsumer(K)$
        $R \leftarrow pull(C)$
        for all $r \in R$ do
            $F \leftarrow getFailedProducers(k)$
            $p \leftarrow getKabisProducer(r)$
            if $p \notin F$ then
                $tp \leftarrow getTopicPartition(r)$
                $q \leftarrow getQueueFor(k, tp, p)$
                $v \leftarrow getValue(r)$
                $push(q, r)$
            end if
        end for
    end if
end for
```

Validator The `Validator`'s responsibility is to map a list of SIDs from the *validation channel* to events on the *storage channel* identified by each SID. This is achieved by the `List<KabisRecord > validate(List<SID>)` procedure, which pseudocode is the following:

Algorithm 2 `validate` procedure

```
$S \leftarrow$ list of signatures passed as argument
$K \leftarrow$ set of *kafka replicas*
$R \leftarrow \emptyset$
for all $s \in S$ do
    $p \leftarrow getKabisProducer(s)$
    $tp \leftarrow getTopicPartition(s)$
    for all $k \in K$ do
        $F \leftarrow$ set of failed producers for *kafka replica* $k$
        if $p \notin F$ then
            $result \leftarrow null$
            $queue \leftarrow getQueueFor(k, tp, p)$
            if $result \neq null$ then
                $wrapper \leftarrow poll(queue)$
                if $signatureVerify(s_v)$ then
                    $result \leftarrow w$
                else
                    $F \leftarrow F \cup \{p\}$
                end if
            else
                $pop(queue)$
            end if
            $r \leftarrow buildKabisRecord(wrapper)$
            $R \leftarrow R \cup \{r\}$
        end if
    end for
end for
```

Given an SID, if its sender was not previously marked as failed, then the first event in that cache is expected to match the SID. Two cases are possible:

- **The event matches the SID** The correct event is found. Pop the not yet inspected queues and return the found event.
- **The SID does not match** Either the sender has failed, or the *Kafka replica* has. Try with the next cache.

To execute a pull, the `KabisConsumer` first pulls from the *validation channel*, getting the new SIDs since the last pull.
As soon as a non-empty SID list is received, this is passed to the `Validator`, which will map each SID to an appropriate `KabisRecord`.
The result of this mapping is finally returned.

6. Performance evaluation

Kabis performance has been empirically evaluated on a setup consisting of 1 `KabisConsumer` and 3 `KabisProducers`. The event service was deployed to be resistant to $F_B = 1$ byzantine failures or $F_C = 3$ crash failures.
Since the experiments were carried out on a single machine, the outcome may be affected by the limited resources and by the absence of network latency.
For each presented combination of event payload and number of validated topics, the setup has been tested by measuring the execution time required to transmit 50000 events from each producer to the single consumer, averaging the results of multiple experiments.
Analysis of each of the systems under continuous load shown that the amount of time and requests needed to reach a stable state was minimal, allowing to take direct measurements of the throughput.
The graphs in figures 4 and 5 summarize the evolution of Kabis consumer and producer throughput with increasing message payload, compared to those of Kafka ans BFT-SMaRt.

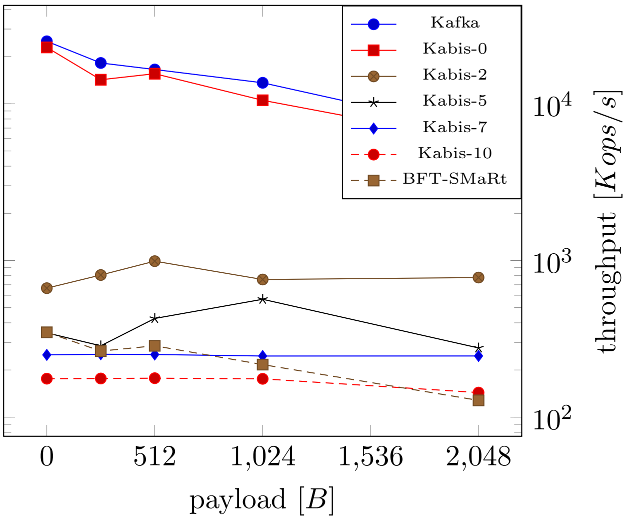

Figure 4: Kabis consumer throughput per payload

The consumer's performance is very close to Kafka's when no validation has been performed. Moreover, when all the topics are validated Kabis' performance is independent from the event payload, allowing it to eventually outperforming BFT-SMaRt.

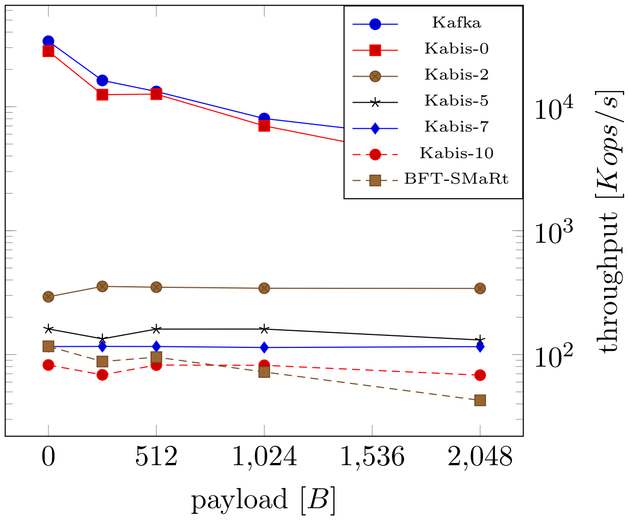

Figure 5: Kabis producer throughput per payload

The producer throughput exhibits the same behavior as the consumer's. The most notable difference is that whenever some topics are validated, the throughput is strongly independent from the payload, suggesting that the producer's throughput is more strictly bounded by that of the underlying validation channel than the consumer's.

7. Conclusions

Existing technologies for event-based architectures can't work in byzantine environments. Even if the user could be able to achieve non-repudiation at the application level, proper byzantine fault tolerance requires the underlying communication protocol to be adjusted.
Kabis has been developed to correctly operate in a byzantine, permissioned environment: offering an API similar to that of Apache Kafka, it can be configured to enrich specific topics with byzantine fault tolerance and non-repudiation, allowing each user to tune the trade-off between performance and received guarantees.
Kabis correctness in presence of byzantine failures has been proven, and experimental evaluation shown the improvement of Kabis over existing byzantine fault tolerant systems, the smaller correlation between its throughput and the payload of events transmitted through it, and its ability to reach performance close to Kafka when it is configured to provide the same level of guarantees.

8. Acknowledgments

First I'd like to thank Professor Alessandro Margara, not only for the guidance he gave me as my advisor, but for being the person who mostly inspired me into joining the Computer Science faculty. Also, I would like to thank Professor Guido Salvaneschi because of his help as my co-advisor. Leon Chemnitz have my gratitude as well for his support in the thesis conceptualization and development.
I would also like to thank Maren Eikerling, Maria Luisa Lorusso, Francesco Vona and Professor Franca Garzotto for our collaboration and the publication of my first research paper.
I'm grateful to all my friends, the ones of a lifetime and the once I made in the last years, for the beautiful time together and the support they gave me along my journey.
Finally my gratitude goes to my family: my parents Sandro and Cinzia and my brother Luca, which love and support allowed me to get through all this work with peace and happiness. Thank you all for your support and inspiration.

References

[1] Amazon AWS. Event Sourcing, 2020.

[2] K Mani Chandy. Event-driven applications: Costs, benefits and design approaches. *Gartner Application Integration and Web Services Summit*, 2006, 2006.

[3] Martin Fowler. Event sourcing, 2005.

[4] Google. Deploying event-sourced systems with cloud spanner, 2020.

[5] Microsoft. Command and query responsibility segregation (cqrs) pattern, 2020.

[6] Joao Sousa and Alysson Bessani. From byzantine consensus to bft state machine replication: A latency-optimal transformation. In *2012 Ninth European Dependable Computing Conference*, pages 37–48. IEEE, 2012.

[7] Ben Stopford. *Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka*. O'Reilly Media, 1st edition, 2018.