POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

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

Tesi di Laurea Magistrale in
Computer Science Engineering - Ingegneria Informatica

Author: **Matteo Secco**

Student ID: 946773
Advisor: Prof. Alessandro Margara
Academic Year: 2022-23

Abstract

Despite their increasing popularity of event-based architecture to develop big scale, highly decoupled services, existing technologies for this kind of architectures rely on the assumption that any process of the system can be trusted.

To get rid of this strict guarantee Kabis has been developed: Kabis is an event-based system resembling the architecture of Apache Kafka, but allowing its users to enforce byzantine trust guarantees on a topic level.

This flexibility allows each user to manage the trade-off between functional guarantees and performance according to its own business requirements. This result has been achieved by deploying Apache Kafka together with BFT-SMaRt, a system allowing event notification in a byzantine environment, and developing a communication protocol which optimizes BFT-SMaRt's usage preserving its trust guarantees.

Kabis has been designed to allow for a smooth migration from a Kafka-based system, since its API is a subset of Kafka's, while exposing optional methods to let the user define which topics should require the trust guarantees provided by BFT-SMaRt.

Experimental evaluation shown that Kabis performance is comparable to that of Kafka when no additional guarantees are provided, and outperforms BFT-SMaRt even when providing its guarantees to all the topics.

Keywords: Event-based system, byzantine fault tolerance, non-repudiation

Abstract in lingua italiana

Nonostante la crescente popolarità delle architetture basate su eventi nello sviluppo di servizi di ampia scala altamente disaccopiati, le tecnologie esistenti per questo tipo di architetture si basano sull'assunzione che ogni processo nel sistema sia fidato.

Per superare questa stringente garanzia è stato sviluppato Kabis: Kabis è un sistema basato su eventi che rispecchia l'architettura di Apache Kafka, ma che permette ai suoi utenti di applicare garanzie di fiducia bizantina a livello di topic.

Questa flessibilità consente a ciascun utente di gestire individualmente il trade-off tra garanzie di fiducia e prestazioni in base ai propri requisiti di business. Questo risultato è stato ottenuto affiancando Apache Kafka a BFT-SMaRt, un sistema che permette la notifica di eventi in ambiente bizantino, e sviluppando un protocollo di comunicazione che permetta di ottimizzare l'utilizzo di BFT-SMaRt mantenendone le garanzie di fiducia.

Kabis è stato progettato per consentire una semplice transizione da un sistema basato su Kafka, dato che la sua API è un sottinsieme di quella di Kafka, fornendo al contempo metodi opzionali che permettono all'utente di definire per quali topic richiedere le garanzie di fiducia offerte da BFT-SMaRt.

Una valutazione sperimentale ha evidenziato che le prestazioni di Kabis sono simili a quelle di Kafka quando le garanzie aggiuntive non sono fornite a nessun topic, e superiori a quelle di BFT.SMaRt anche quando queste garanzie sono fornite a tutti i topic del sistema.

Parole chiave: Sistema a eventi, tolleranza ai guasti bizantina, non-ripudio

Contents

Introduction

In event-based interaction, processes execution depends on events that happened into the system. An event can be defined as "a significant change in state" [12]. When a process becomes aware of an event that happened, it may react to it by executing some logic, which may result in some other event happening for the process. This is called *handling an event*, while the technology used to make a process aware of events happened to other processes is called *event notification*. Therefore, an event-based system is a system where processes communicate between each other through event notification, and their execution is limited to event handling.

To propagate state changes across services, event sourcing [24] 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 [43], Amazon AWS [4], and Google Cloud [25]. Event-sourcing architectures have the benefit of allowing client processes to only read from the log when they are computationally able of handling a new event.

However the system used to store the history of events may become a single point of failure for the entire architecture and therefore needs to be replicated. Existing technologies are well suited for the simpler failure models, where a machine or a process simply stops operating but otherwise behave as they are expected. But it is common for service-based systems to cross the boundaries of individual business units or even organizations. In this setting, the parties involved in the communication might not fully trust each other and the events they notify. The event log in an event sourcing application provides evidence for the responsibility of each component, ensuring that each party can be held accountable for its actions. Unfortunately, existing solutions for event sourcing inevitably require to trust a third-party authority that stores the event log (which may collude with some of the parties) or let some of the parties to store it itself. Even if all of the parties behave correctly, the processes they own may be compromised by an attacker and start behaving

maliciously, hindering accountability in the whole system.

This imposes to consider the infrastructure supporting the system to be able to correctly operate even when some of the processes that run it may act maliciously. Technologies addressing such behaviors exist, but their use is limited due to their high performance cost and its strong dependence on the payload size of the event notifications exchanged. 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.

This thesis is divided as follows: chapter 1 describes the framework Kabis operates within, introducing the terminology from distributed systems theory that will be needed in the rest of the document and presenting the technologies used as performance reference and for the system implementation. Chapter 2 describes Kabis APIs and compares them to those of Apache Kafka, also presenting the fault-tolerance and trust guarantees the system can provide. Chapter 3 describes the system implementation, describing in detail the components Kabis is composed of and their correct deployment among different parties. Chapter 4 finally compares the performance of Kabis with those of the reference technologies, providing interpretation of the measured performance and proposing possible optimizations to improve over the achieved results. Chapter 6 will draw the conclusion and present some possible future developments of this work.

1 | Background

This section will introduce some concepts and terminology from the literature, and present the technologies adopted during this work. In section 1.1 the concept of event-based architecture is presented together with the kind of processes generally composing such architecture, the API it offers, and some variations existing in the literature. Section 1.2 describes the failure models adopted in this study, and section 1.3 the models for event ordering. Section 1.4 presents the concept of *non-repudiation*, an high-level requirement in computer security which is one of the goals of this work. Finally, sections 1.5 and 1.6 present two technologies upon which the developed system is based.

1.1. Event-based architectures

Different configurations of event-based architectures exist in the literature. They mainly differentiate by the way clients can select which events to receive, though they all share some common terminology and the communication architecture is mostly standard. As shown in figure 1.1, the architecture is composed by three components:

Producers are processes publishing events into the system. They can also be referred to as *publishers*.

Consumers are processes subscribing for events and getting notified for new events. They are also called *subscribers*.

Event-service is the central service coordinating the communication. Producers publish events to this service, and consumers subscribe and unsubscribe to this service as well. Also, this service notifies consumers of new event they are subscribed for.

And offers four communication primitives:

subscribe Can be used by a consumer to inform the event service about the events it wants to be notified about. The subscription logic is the main difference among different kinds of event-based architectures.

unsubscribe Allows a consumer to revoke a previously issued subscription.

publish is used by producers to send new events to the consumers

notify is used by the event-service once an event is published, to notify it to all the consumer which subscription matches the event.

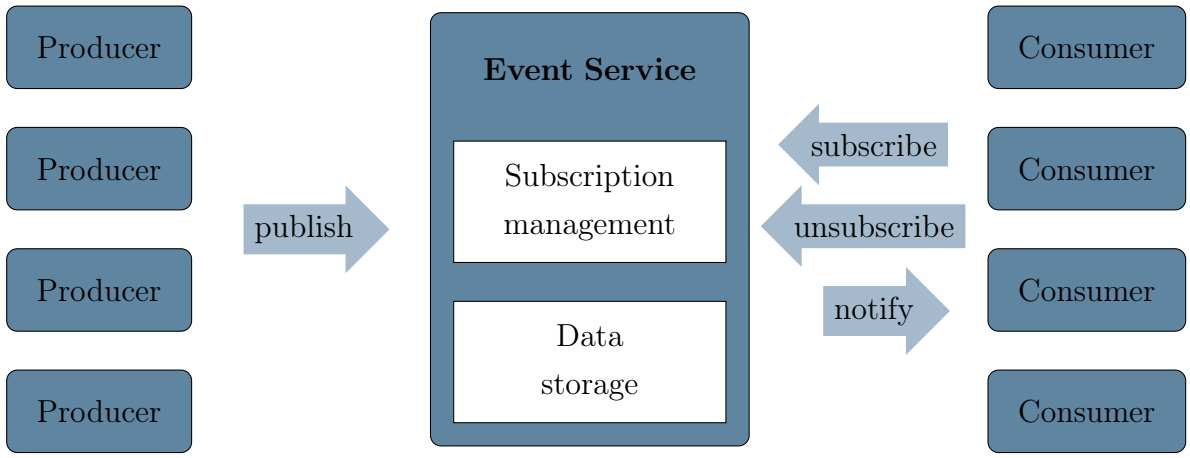

Figure 1.1: Event-based architecture

Different subscription logic define different kind of event-based architectures. The main architectures are:

Topic-based Events are classified in *topics* to which consumers can subscribe. In basic topic-based architectures, consumers subscribe to a topic by its name; however more flexible forms of subscription are possible (for example allowing to use wildcards or regular expressions to match the topic name).

Content-based Subscription is issued based on the event content. The consumers gain more fine-grained control of which events to receive, being allowed to provide predicates on the event during the subscription.

Type-based This is an extension of the topic-based model, where an event type is used in place of a topic name, allowing type-checking at compile time.

This thesis focuses on topic-based systems. In real deployment, the *event service* is usually a distributed component. Therefore topics are usually replicated for fault tolerance, and partitioned for performance (allowing to split the workload among different producers and consumers).

1.2. Fault tolerance models

Focusing on the message-driven architecture used to notify event, its correctness is a mandatory requirement for the correctness of any system built on top of it. The messaging infrastructure may be composed by multiple internal processes, and the whole architecture should keep operating even in when some of these processes become unavailable (which is called a crash failure). Moreover, the infrastructure should continue to operate correctly even when some of its processes are exhibiting arbitrary, possibly malicious behavior (byzantine failure). Formally:

Crash failure happens when a process P takes an infinite amount of time to deliver a message, and then stops delivering messages.

Byzantine failure happens when process P exhibits arbitrary, possibly unexpected, behavior. It was formalized for the first time as a mathematical problem known as "The byzantine generals problem" [37].

System can be developed to be resistant to a certain number of failures of either the crash or the byzantine kind. In particular:

Crash fault tolerance (CFT) is the ability of a system to operate correctly when some of its processes exhibit crash failures. This can easily be achieved with replication, replicating each process F_C+1 times across different machines in order to be resilient to up to F_C brokers crashing; the actual number of replicas is usually determined at deploy time. The resulting system is said to have a crash fault tolerance (CFT) up to F_C failures.

Byzantine fault tolerance (BFT) is the ability to operate correctly in presence of byzantine failure. Byzantine fault tolerance (BFT) requires a minimum of $3F_B + 1$ replicas to cope with F_B failures.

In an optimal configuration, a system will always be more resistant to crash failures than to byzantine ones: inverting the equations above, a system composed of N replicas will be able to tolerate $F_B = \left\lfloor \frac{N-1}{3} \right\rfloor$ byzantine failures and $F_C = N - 1$ failures. It is evident that $F_B < F_C$ as long as soon as $N > 1$, which happens if some replication is in place.

1.3. Ordering models

As already introduced, ordering the notification of events is crucial in an event-based system for correctness: events being notified out of order could make the internal state of the processes involved into the system inconsistent among each other. An event system guarantees an order of notification of events if the events are always notified to processes in a sequence that is compatible to that order. A message-driven architecture guarantees an order of delivery of messages if all the messages are delivered in a compatible sequence. If an event-based system exploits a message-driven architecture for event notification, and processes handle events as soon as their representation is received in the form of a message, then any ordering guarantee for the messages is inherited to event handling. For this reason, the ordering model we present to and any following reasoning about order will be carried out on a messaging level instead of an event level. Two ordering properties are of interest for this study: FIFO order, total order:

FIFO order is an ordering requirement that guarantees that messages from a process will be delivered in the same order they were sent. Messages from different processes can however be delivered in any order. Formally, a message-driven architecture is said to be FIFO-ordered if for each process p in the system and any two messages m_1 and m_2 sent by p, if p sent m_1 before m_2, then for any message q in the system it does never happed that q reads m_2 before m_1.

Total order is a mathematical definition of order in which any two elements are comparable. In a message-driven architecture, this means that for any couple of distinct messages m_1 and m_2, all correct processes will either receive m_1 always before m_2, or m_2 always before m_1.

In general, total order does not imply FIFO order, nor FIFO order implies total order: a system may ensure FIFO order and not total order, the opposite, both of them or neither. For our use-case an ordering that is both FIFO and total is required for the correctness of the execution.

1.3.1. Consensus

Consensus is a mechanism by which a set of replicas try to reach an agreement over some value. Consensus provides multiple benefits: it can be used to provide total order without a relying on central authority (by having all the replicas to agree on the order itself) and, used in combination of a digital signature algorithm, can be used to achieve non-repudiation not only over the content of any shared message, but also over the order

of the messages itself.

In a system only affected by crash failures, consensus can be reached using $F_C + 1$ replicas and $F_C + 1$ communication rounds to tolerate up to F_C failures, using algorithms such as *FloodSet* [41]. In more realistic scenarios that contemplate partitioning of the network, algorithms such as *Paxos* and *Raft*, can reach consensus with $2F_C + 1$ non-byzantine replicas [29, 36, 48]. In presence of byzantine failures instead, it can be proved that atomic reliable broadcast and the consensus problem are equivalent [15, 26, 44]. Providing byzantine fault tolerance is then enough to guarantee consensus on both the content and the order of messages, and it requires requires $3F_B + 1$ replicas in presence of F_B failures.

Permissioned environment A system is said to operate in a permissioned environment if any participant of the system can be uniquely identified by other participants any time it operates into the system. Permissioned environments are typically created through digital signatures, having each participant to share its public key on a trusted channel before entering the system.

1.4. Non-repudiation

In the field of cybersecurity, *non-repudiation* is defined as "Assurance that the sender of information is provided with proof of delivery and the recipient is provided with proof of the sender's identity, so neither can later deny having processed the information" [56]. This property is of great legal interest as it allows to stipulate contracts among parties that don't fully trust each other.

Digital signature Non-repudiation is easily achieved through digital signatures, a mathematical technique to ensure authenticity and integrity of a message. Authenticity is a property that allows the receiver of a message to know the sender, while integrity ensures that the content of the message has not been altered during transmission. In this work, digital signatures are created by using the *ECDSA* asymmetric encryption algorithm to sign hashes obtained through the *SHA-256* algorithm.

1.5. Apache Kafka

Apache Kafka is "an open-source distributed event streaming platform" [2] offering an excellent degree of scalability, high throughput and low latency, persistency of the history of messages, and a flexible and configurable crash fault tolerance. These characteristics made Kafka a de-facto standard for event-based architectures.

In Kafka, an event is characterized by a *key*, a *value*, a *timestamp* and optional *metadata headers*. Client processes are divided between *producers* and *consumers* the first are processes that publish events, while the seconds receives them. Producers and consumers are strongly decoupled.

Events are organized and persisted in *topics*, an abstraction to aggregate events with the same semantic meaning. Topics are always multi-producer and multi-consumer: they can be written on by any number of producers and read form by any number of consumers.

Topics are then *partitioned*, allowing to divide the physical allocation of data among multiple locations to achieve an high degree of scalability. Kafka guarantees that any consumer will always read messages from a given topic-partition in the same order (topic partitions are totally ordered). Multiple consumers can cooperate to consume messages from a topic by being part of a same *consumer group*. Kafka will dynamically divide the partitions of a same topic among consumers of the same group, splitting the workload among them.

To achieve crash fault tolerance, each topic can be replicated, resulting in each of its partitions to be replicated. The number of replicas (and the consequent number of crash failures tolerated) is configured by the system administrator and can be changed at run-time.

To manage crash recovery, the *offset* of the last message read by a *consumer* is stored in Kafka itself (using an internal topic). This ensures that if a *consumer* fails and recovers later, it can restore its *offset* and resume reading from where it left. *Consumers* can also arbitrarily change their offsets, allowing to skip some events or reading them multiple times.

A service infrastructure is required to achieve this: topic-partitions are hosted, persisted and replicated on *brokers*, specialized processes responsible to store the events sent in the system in a log and provide them upon requests. Brokers communicate with client processes on demand, and also communicate between each other directly (for example to propagate writes to different replicas or to rebalance the workload among them). Finally, brokers need to be coordinated by an external service. In the past this could be done only using Apache Zookeeper. Despite nowadays KRaft can also be used in place of Zookeeper, the latter is the adopted technology for this study.

The offered guarantees depends on the configuration of the system: Kafka always provides FIFO and total order on a partition level, as well as CFT. Nevertheless, it does not provide BFT, and all the guarantees it provides do not hold in presence of byzantine failures and

does not ensure non-repudiation[1].

As a result of its popularity and its good results in terms of provided guarantees, Kafka is both part of the Kabis system and one of the reference technologies to measure its performances.

1.6. BFT-SMaRt

In section 1.5 is highlighted how Kafka can be used to ensure FIFO-total order and crash fault tolerance. In the scenario just presented however, stronger assumptions are required: byzantine fault tolerance is mandatory to handle possibly untrusted participants, total FIFO order is required to ensure consistency among the state updates of different processes, and consensus is needed to ensure non-repudiation. A stronger technology is now presented, which can meet all of these functional requirements.

BFT-SMaRt is an open-source library providing byzantine fault tolerance product of active research, currently maintained and extensively optimized [13]. We used this library both as part of Kabis and as a reference technology for existing BFT systems. BFT-SMaRt operates in a permissioned, byzantine environment, uses a custom protocol called MOD-SMaRt to reach consensus [53] and implements different optimization to speedup fault recovery [7] and operation throughput [8].

VP-Consensus BFT-SMaRt uses a stronger definition of consensus called *Validated and Provable Consensus (VP-Consensus)*. This property allows the concept of *non-repudiation* to be extended, making not only the single message non-repudiable but also the order in which it was received.

- *Validated* means the protocol receives a predicate together with the proposed value to agree on, which any decided value should satisfy.
- *Provable* means that the protocol generates a cryptographic proof that certifies that value v was decided in consensus instance i.

BFT-SMaRt exposed two APIs: the *service replica* and the *service proxy*. A *service replica* is a process involved in the BFT protocol which can process commands (represented as a `byte[]`). The logic to serialize, deserialize and process the commands is left to the user of the framework. Different logic may be defined if the client asks to process a command in a total ordered way, or without ordering guarantees. A *service proxy* is just a client

[1]For example, a single byzantine broker may send the messages to clients in any order.

process which interacts with *service replicas*, requesting the execution of commands either in total order or in any order.

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 deploy-time or at run-time, according to their business requirements (as described in section 2.1). The choice to mimic Kafka architecture as much as possible is to allow users to easily migrate from a Kafka system to a Kabis system in any use case.

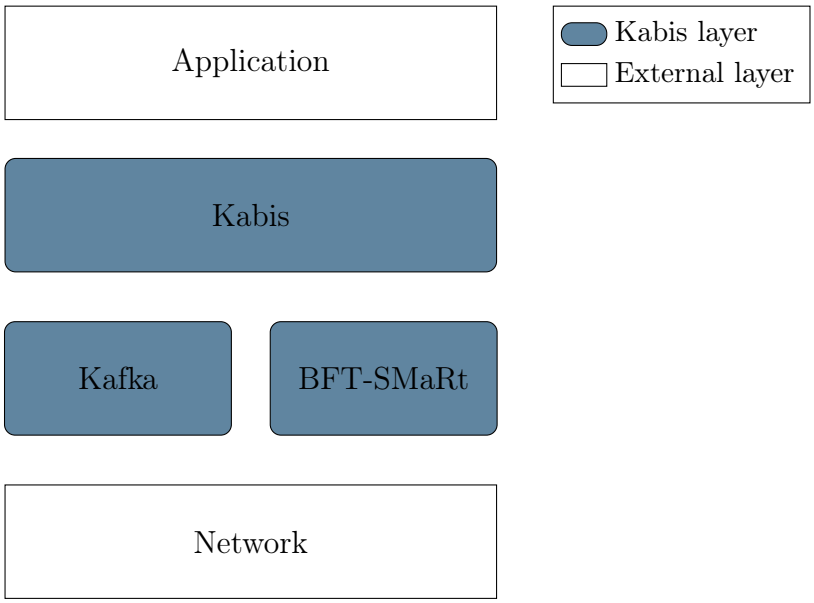

Figure 2.1: Kabis layer stack

2.1. Functional guarantees

Kabis offers to its users the ability to decide to pay an additional performance overhead to seek extended consistency guarantees. Kabis is in fact able to offer two sets of guarantees, described later in this section. However, due to this level of reconfigurability, which set of guarantees will be obtained when an event is consumed depends on the combined configuration of both the producer of an event and the consumer consuming it: only if

both of them payed the extra overhead, then the event will be delivered with the stronger set of guarantees. Otherwise, the delivery will only benefit of the weaker set.

For each existing topic, a producer can be configured to publish events on that topic as *validable* events (by paying the overhead price) or *unvalidable* events. In the first case, consumers will be able to validate the events published from this consumer to get the extended level of guarantees that Kabis can produce. In the second case, a consumer will have no way to validate them and could either deliver it with the basic set of guarantees or discard it.

From the consumer perspective, when it is configured to not validate the events it will immediately notify each event received to the application level. If the consumer is configured to only accept validated events, it will deliver only validable events that pass the validation (the validation process is explained in section 3.3.2). Upon receiving a not validable event, it will drop it. The provided level of guarantees according to the combination of producer and consumer configuration is described in table 2.1.

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

Table 2.1: Class of guarantees by producer's and consumer's behavior

2.1.1. Basic guarantees

Kabis *basic guarantees* are the very same of Kafka:

FIFO order on partition level For any topic-partition n and producer P, if P published events e_1 and e_2 on n in this order, than to no consumer will be notified e_2 before e_1.

Total order on a partition level For any topic-partition n, the messages in that partition are totally ordered. Any consumer will receive events from n in the same order.

Crash fault tolerance The system can continue to work and provide the ordering guarantees presented above even in presence of crash failures, as long as the system is deployed with a sufficient number of replicas. How many replicas for each kind of process in the system are required for this kind of fault tolerance is described in section 3.1.1.

Since the system in this configuration is only CFT and not BFT, all of its guarantees do not hold any longer in presence of byzantine failures. If the user requires byzantine resilience, it should use the extended set.

2.1.2. Extended guarantees

Kabis *extended guarantees* include all of the basic guarantees. Moreover, this set of guarantees also includes:

Byzantine fault tolerance The system can continue to work and provide the consistency guarantees presented above even in presence of byzantine failures, as long as the system is deployed with a sufficient number of replicas[1]. How many replicas for each kind of process in the system are required for this kind of fault tolerance is described in section 3.1.2.

Non-repudiation An event sent through Kabis can only be notified after all the (correct) parties (including the sender) reached consensus on its content and the order of notification. The reached consensus and the agreement of each party can be verified cryptographically at any future time, making it impossible for anyone to repudiate it.

2.1.3. Update topology API

The *UpdateTopology* API allows users to configure which *topics* will be validated and which won't. The API only consists of the `void updateTopology(Collection<String>)` method. Conceptually, this method splits the set of existing topics in two: the ones that will be validated, and the one who don't. This classification is called a *topology*, and is local to each producer or consumer process. The caller updates the topology specifying the topics to be validated: the remaining one will automatically be considered not to validate. This method is not incremental: calling it erases the previously configured topology replacing it with the new one.

2.2. Kabis event service

As already introduced, we wanted Kabis architecture to be as similar as possible to the one of Kafka described in section 1.5. For this reason, Kabis inherits Kafka's architecture of a partitioned, persisted, replicated topics. Client processes are divided into two

[1] As it will be shown in section 3.1.2, the extended guarantee set provides CFT at a different cost that the basic guarantee set (requiring a different number of replicas to tolerate the same number of failures).

kind: `KabisConsumer` and `KabisProducer`, the first are processes capable of consuming events from the topics, while the second can only publish events on them. `KabisConsumers` and `KabisProducers` are strongly decoupled.

`KabisProducers` can publish events to any topic, and a single producer can publish to multiple topics. `KabisConsumers` can consume messages from multiple topics as well. Moreover, for the sake of load balancing, consumers can be grouped into *consumer groups* to split the workload.

2.2.1. KabisRecord

Kafka uses different data objects in its APIs. The `Consumer` API returns events in the form of `ConsumerRecord` objects, while the `Producer` API receives events as instances of the `ProducerRecord` class. Kabis only uses a single class to represent events: the `KabisRecord` class.

```
1  public class KabisRecord<K, V> {
2    private final String topic;
3    private final int partition;
4    private final K key;
5    private final V value;
6
7    private final byte[] signature;
8
9    public KabisRecord(String topic,
10                      int partition,
11                      byte[] signature,
12                      K key,
13                      V value)
14   {/* assignments */}
15
16   public KabisRecord(String topic, K key, V value) {
17     this(topic,-1,null,key,value);
18   }
19 }
```

As shown by the code above, a `KabisRecord` object has 5 attributes: `topic`, `partition`, `key`, `value`, `signature`. The first 4 attributes have the same semantic meaning of Kafka's, while the `signature` is specific to Kabis. This attribute contains the digital signature of the rest of the record, computed by the `KafkaProducer` that originally sent the record to allow `KabisConsumers` to verify the identity of the sender and the integrity of the event representation. This field can be empty if the record represents an *unvalidable event* (see

section 2.1).

2.2.2. 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, containing methods used to make reads and writes transactional (this section is typically used when a `Producer` is writing the results of processing events read from a `Consumer`).
2. Metadata gathering, which are used to retrieve information about the system's deployment and the `Producer` itself.
3. Sending methods, used to publish events into the system.
4. Disposing methods, used for graceful termination.

KabisProducer API hides all the transaction management from the user/system administrator and handles it internally, while all the other clusters could be implemented into Kabis without impacting the system's design. This means that at the present time, `KabisProducer` API consists of most of the methods of groups 3 and 4 of Kafka's `Producer` API, but a 1-to-1 correspondence between the methods of clusters 2, 3 and 4 is theoretically possible. The correspondence between Kafka and Kabis implemented methods is shown in table 2.2. The core method of this API is `void push(KabisRecord)`, that allows to publish an event through Kabis. Its implementation will be discussed in section 3.3.1.

	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 2.2: Producer API of Kafka and Kabis

2.2.3. KabisConsumer API

As for the `KabisProducer` API, the `KabisConsumer` API consists of a subset of Kafka `Consumer` API. Because of the higher complexity of the API, the number of clusters is doubled:

1. Metadata gathering, used to retrieve information about the system's current deployment and the `Consumer` itself.
2. Topic navigation, used to change the `Consumer`'s offset to arbitrary move the reading position on the topic-partition.
3. Execution suspension, which allows to pause the `Consumer`'s activity (influencing the behavior of pull operations).
4. Offset commit, which is crucial for error recovery, as it allows the `Consumer` to resume reading from where it was before crashing.
5. Rebalance. This cluster is composed by a single method, which purpose is to exists in Kafka to rebalance the `Consumer` group in extraordinary circumstances dictated by business logic.
6. Subscription management contains methods used to instruct the `Consumer` about

which topics to read from. In Kafka, this can be done either on a topic level (leaving the partition assignment to the infrastructure), or manually at partition level. The two methods cannot be used interchangeably.

7. Poll, a single method used to fetch new data.
8. Disposing methods, to gracefully stop the `Consumer`.

KabisConsumer API only expose methods from clusters 6, 7 and 8. Metadata gathering, execution suspension, and rebalance methods (clusters 1, 3, 5) could easily be added to the system, while the other clusters have been left out by design. Specifically, topic navigation (cluster 2) should be prevented in order to enforce a reading order, offset commit (cluster 4) should be managed by Kabis internally to be able to provide the extended set of guarantees described in section 2.1.2. Correspondence between Kafka and Kabis methods is presented in table 2.3. At the core of the `KabisConsumer` API is the `Iterable< KabisRecord> pull(Duration)` method, used to read from the system all the records not yet read (from the topics to which the `KabisConsumer` is configured to read).

	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 2.3: Consumer API of Kafka and Kabis

3 | System design and implementation

Kabis is designed as two independent communication channels accessible from `KabisProducers` and `KabisConsumers` as represented in figure 3.1. The *storage channel* is used for basic communication and contains the topics in a readable format (events are serialized by the producers and deserialized by the consumers). The *validation channel* can instead optionally used to provide the extended guarantees to the event notifications. It does not contain the full description of events but *Secure identifiers (SID)*, obtained through digital signature. This allows `KabisConsumers` to verify the correspondence of an event notification on the *storage channel* with its SID on the *validation channel*, while keeping the bandwidth on the *validation channel* constrained to a constant value.

Client processes are the `KabisProducers` and `KabisConsumers`. `KabisProducers` always publish events on the *storage channel*, and can optionally also write their SID them on the *validation channel*. Equally, `KabisConsumers` always receive notifications from the *storage channel* and can optionally also read from the *validation channel*.

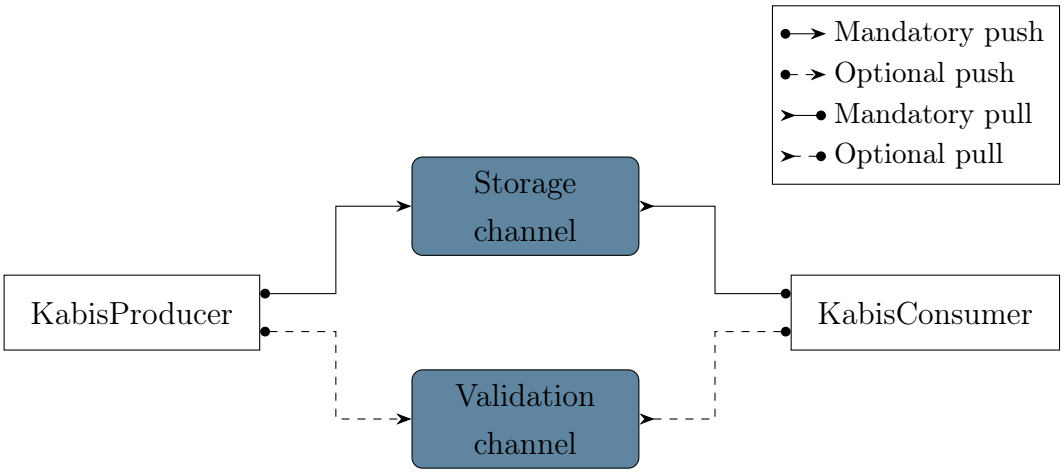

Figure 3.1: Kabis components.

3.1. Component implementation

This section describes the actual implementation of the components shown in figure 3.1, which sub-components and/or processes constitute them, and how many replicas are required for each type of component, sub-component and process. The actual deployment of the components, i.e. where each component is hosted and how it is distributed, is described in section 3.2.

3.1.1. Storage implementation

The *storage channel* is developed as a replicated Kafka infrastructure. Events are published through this channel after being wrapped in the `MessageWrapper<V>` class:

```
1 public class MessageWrapper<V> {
2   private final V value;
3   private final Long senderId;
4 }
```

The class enriches the original event representation (which is the attribute `value`) with a `senderId` used by receivers to order the received `MessageWrappers` For *unvalidable events*, only the `value` is relevant. The other attributes are set to `null` to optimize bandwidth consumption.

The term *Kafka replica* will be used to refer to one of these Kafka infrastructures, and in the rest of the thesis it will generally be treated as a black-box system. A single *Kafka replica* is composed by $F_C + 1$ Zookeeper instances and $F_C + 1$ Kafka brokers, where F_C is the number of crash failures we want to tolerate in the system with the basic guarantees set. This number ensures that at least one Zookeeper instance and one Kafka broker will always be working.

Then, in order to tolerate F_B byzantine failures, we want to have a total of $F_B + 1$ independent *Kafka replicas* in our system. This is not in contradiction with the lower bound on the number of replicas presented in section 1.2: this is because each *Kafka replica* here is a black box system independent from the others, each with a byzantine fault tolerance to 0 failures. As described in detail in sections 3.3.1 and 3.3.2, to achieve BFT it is only needed that at least one of the *Kafka replicas* of the *storage channel* is correct. In presence of F_B failures and $F_B + 1$ replicas, at least one replica cannot be failing.

Summing up, the number of individual process required to deploy a *storage layer* resistant

to F_C crash failures and F_B byzantine failures is $(F_C + 1)(F_B + 1)$ Zookeeper instances and $(F_C + 1)(F_B + 1)$ Kafka brokers, for a total of $2(F_C + 1)(F_B + 1)$. The resulting architecture is represented in figure 3.2.

Client processes access *Kafka replicas* using the standard Kafka APIs, and it is responsibility of the `KabisProducer` to replicate their operations to all of the *Kafka replicas*. A `KabisProducer` failing to do so is considered to be failing in a byzantine way.[1]

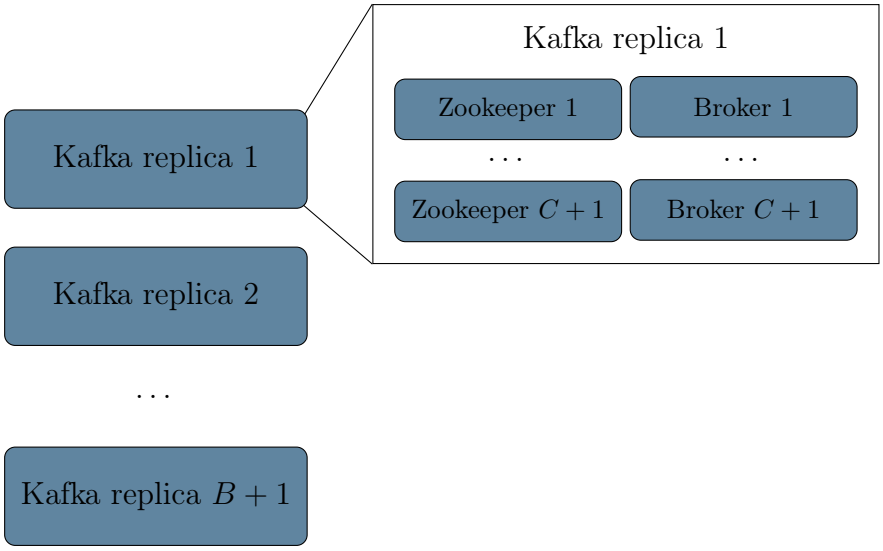

Figure 3.2: Kabis storage channel for B byzantine and C crash faults.

3.1.2. Validation implementation

The *validation channel* is a straightforward implementation of a BFT-SMaRt system introduced in section 1.6, with $3F_B + 1$ *service replicas* required to tolerate F_B byzantine failures. Each party using the system will own exactly one *service replica*, to ensure fairness and avoid concurrent accesses to the *validation channels* from the same party.

Data is sent through this channel as a `SecureIdentifier` (SID) object:

```
1 public class SecureIdentifier{
2   private byte[] signature;
3   private String topic;
4   private int partition;
5   private int senderId;
6 }
```

An SID contains the following attributes:

[1] For `KabisProducers` producing validable events. `KabisProducers` only aiming for the basic set of guarantees can in fact only use the first *Kafka replica*, as described in sections 3.3.1 and 3.3.2.

- `signature` is a digital signature obtained by applying the *SHA-256* algorithm first and the *ECDSA* cryptography algorithm later to the serialized representation of an event's `key`,`value`,`topic`,`partition` and `senderId`.
- `topic` the topic to which the identified event belongs.
- `partition` the Kafka's topic-partition to which the identified event was assigned.
- `senderId` identifies the *producer* process that generated the identified event.

Despite each user only owning a single *service replica*, the application workload can still be split among distinct processes. Client processes, in fact, will not directly interact with the *service replica* but will instead use a *service proxy* to interface with the *validation channel*. The BFT-SMaRt library that we used to implement the *validation channel* does guarantee BFT sequentially ordered communication among the *service replicas*. Since an additional layer of communication exists between the *service replica* and the *service proxy*, we had to prove that such guarantees are maintained in our use case. This is done in sections 3.3.1 and 3.3.2.

3.1.3. Producer implementation

Each `KabisProducer` has access to both the *storage* and the *validation channel*. This is obtained by providing each Kabis `KabisProducer` with a distinct `KafkaProducer` for each *Kafka replica* of the *storage channel*. Moreover, the producer will have a *service proxy* to interface with the *validation channel*. This allows a party to run multiple `KabisProducers`.

When a `KabisProducer` is sending an *unvalidable event*, it will only publish events to the first *Kafka replica*. This is enough to provide the basic set of guarantees described in section 2.1.1. To send a *validable event* instead, the producer will have to send it to all the *Kafka replicas*, and through the *service proxy* as well.

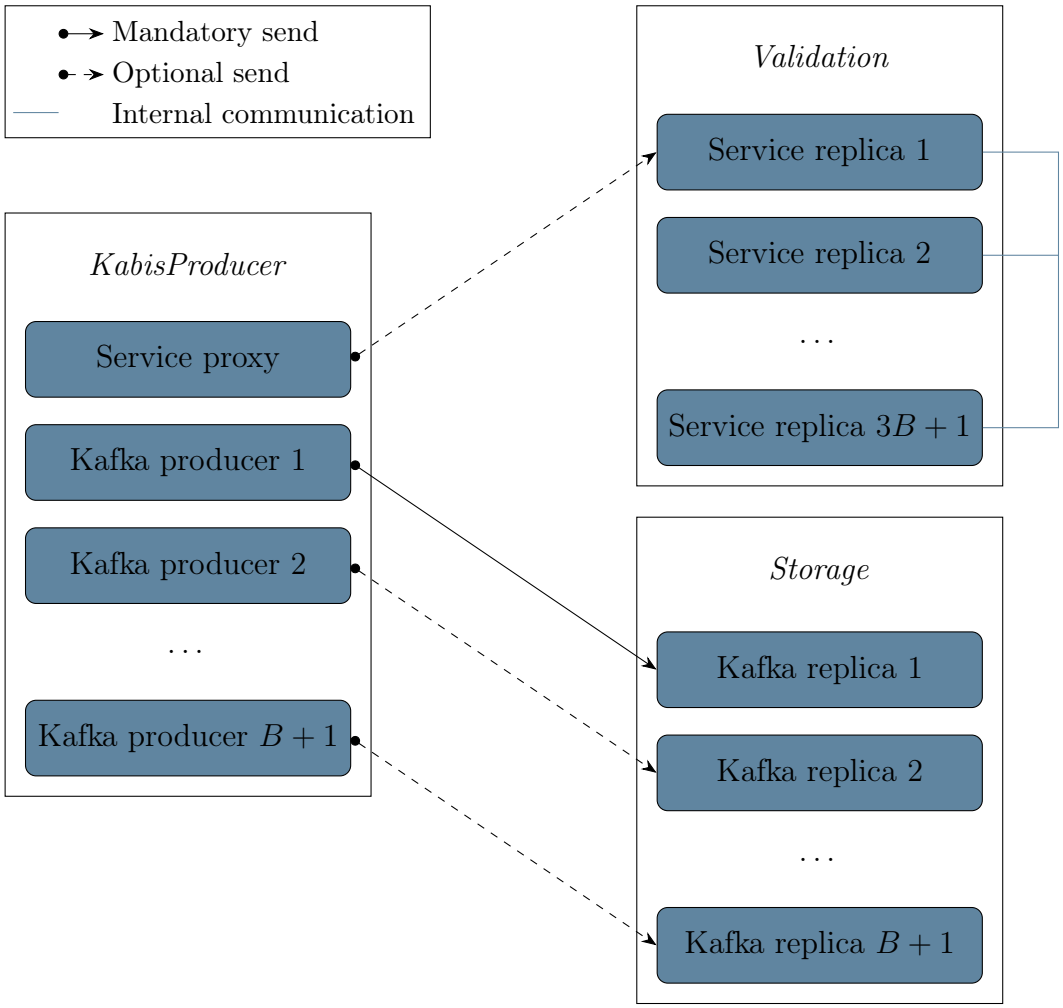

Figure 3.3: *Producer* design.

3.1.4. Consumer implementation

A `KabisConsumer` is a component that receives event notifications from the system. `KabisConsumer` 's design is specular to that of the `KabisProducer`: it is equipped with a `KafkaConsumer` for each *Kafka replica* and a*service proxy* to interface with the *validation channel*.

To consume events without validating them, a `KabisConsumer` can just read them from the first *Kafka replica*. To be able to validate an event and get the extended set of guarantees however, the `KabisConsumer` will have to pull from all the *Kafka replicas* as well as the *service proxy*.

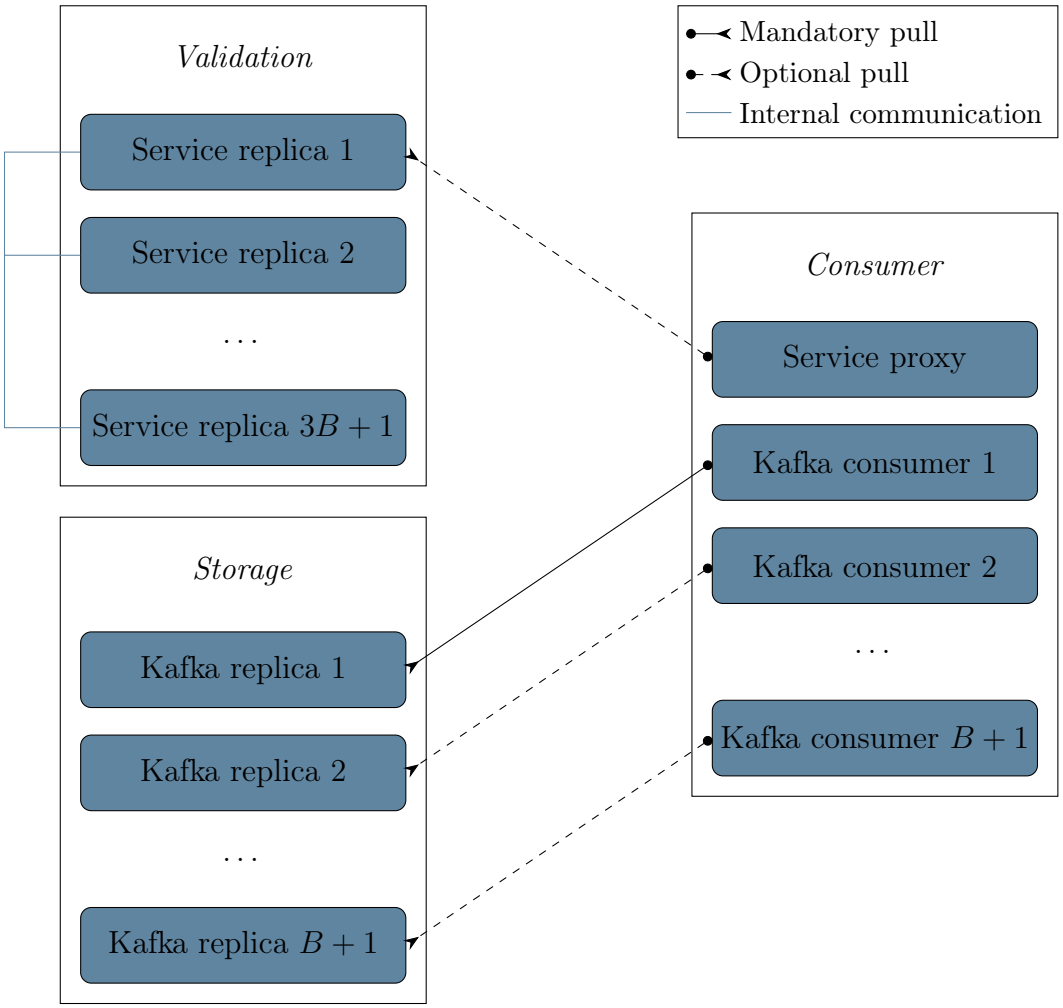

Figure 3.4: *Consumer* design.

3.2. Deployment

The ideal use case for Kabis is when single parties cannot trust each other. In such a case, to use Kabis effectively, it would be necessary that each party hosts one *service replica*, independently from the number of `KabisProducers` and `KabisConsumers` the party will deploy. This makes the *validation channel* perfectly distributed among the parties. The deployment of the *validation* component is exemplified in figure 3.5.

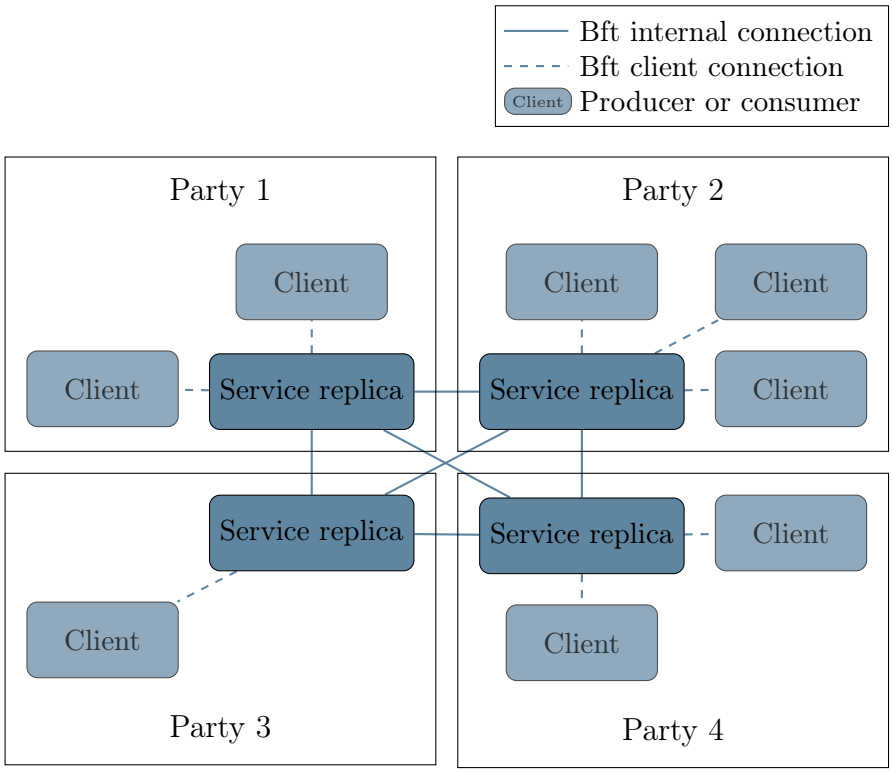

Figure 3.5: *Validation* component distributed among 4 parties, tolerant to at most 1 byzantine failure or 3 crash failures.

With regard to the *storage channel*, the deployment is less rigid. If the number of parties using the system is N, then the maximum amount of byzantine failures that the system will be able to tolerate is $F_C = \lfloor \frac{N-1}{3} \rfloor$. A correct deployment of the *storage channel* will then require $F_C + 1 = \lfloor \frac{N-1}{3} \rfloor + 1$ *Kafka replicas* (as described in section 3.1.1). Parties may want to host a *Kafka replica* each for further safety (to avoid some of them hosting a replica and some not, which may give the impression of a cost or responsibility imbalance[2]. Even if this is actually possible, the same level of equity can be achieved with less processes.

The maximum amount of crash failures tolerable from a deployment with N parties is $F_C = N - 1$. This means that to maximize the CFT of the system, each *Kafka replica* should be internally replicated $F_C + 1 = N$ times (resulting in N Zookeeper instances and N Kafka brokers for each *Kafka replica*). This allows to perfectly distribute each *Kafka replica* among all the existing parties (each party having exactly 1 Zookeeper instance and 1 Kafka broker for each *Kafka replica*), avoiding some of them to be in a possibly privileged position. In figure 3.6 an example deployment of the *storage channel* is shown,

[2]Only some parties would carry the cost of hosting a *Kafka replica*, however those parties could maliciously trigger byzantine failures easier than those who don't host any replica.

which is able to support the *validation channel* deployment of figure 3.5.

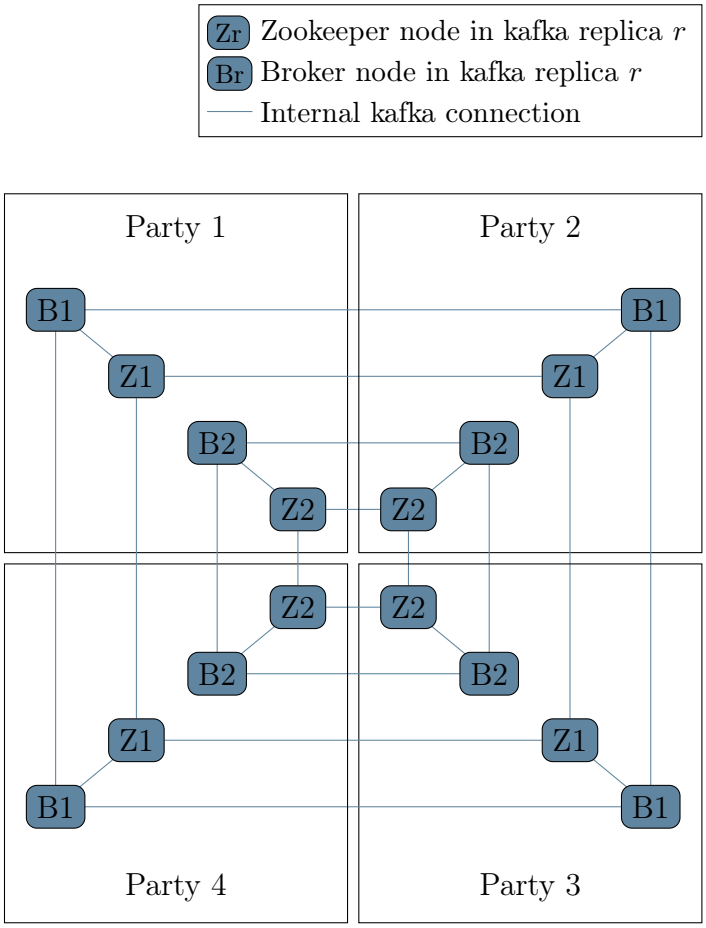

Figure 3.6: *Storage channel* distributed among 4 parties, tolerant to at most 1 byzantine failure or 3 crash failures.

3.3. Network protocols

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

3.3.1. Push protocol

`KabisProducers` publishes events using the `void push(KabisRecord)` primitive. 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.

As shown in the diagram in figure 3.7 (where the *storage processing* and *validation processing* boxes are used to collapse the internal behavior of the communication channels and will be expanded later), the application layer calls the push method passing the `KabisRecord` to be pushed into the system. This record will have empty `signature` and `partition`, since those attributes can only be set by the producer. Upon receiving the record, the `KabisProducer` first computes the partition and the signature for the record. The logic to compute the partition is provided by the system administrator by implementing the `Partitioner` interface provided by Kafka, the same way a Kafka producer would do. The signature is computed using the *SHA-256* and *ECDSA* algorithms, where the payload is obtained from the concatenation of the topic, partition, key, value.

As the SID of the event is ready it is used to create a `MessageWrapper`. Finally, in parallel calls, the `KabisProducer` uses its `KafkaProducers` to sent the `MessageWrapper` to each *Kafka replica* of the *storage channel*, and its *service proxy* to send the SID to the *validation channel*.

The channels process the received data (as described below) and notify the `KabisProducer` upon completion. At this point, the `KabisProducer` returns to the caller.

In figure 3.7 is represented the just described interactions, hiding the internal operations of the *storage* and *validation channel*.

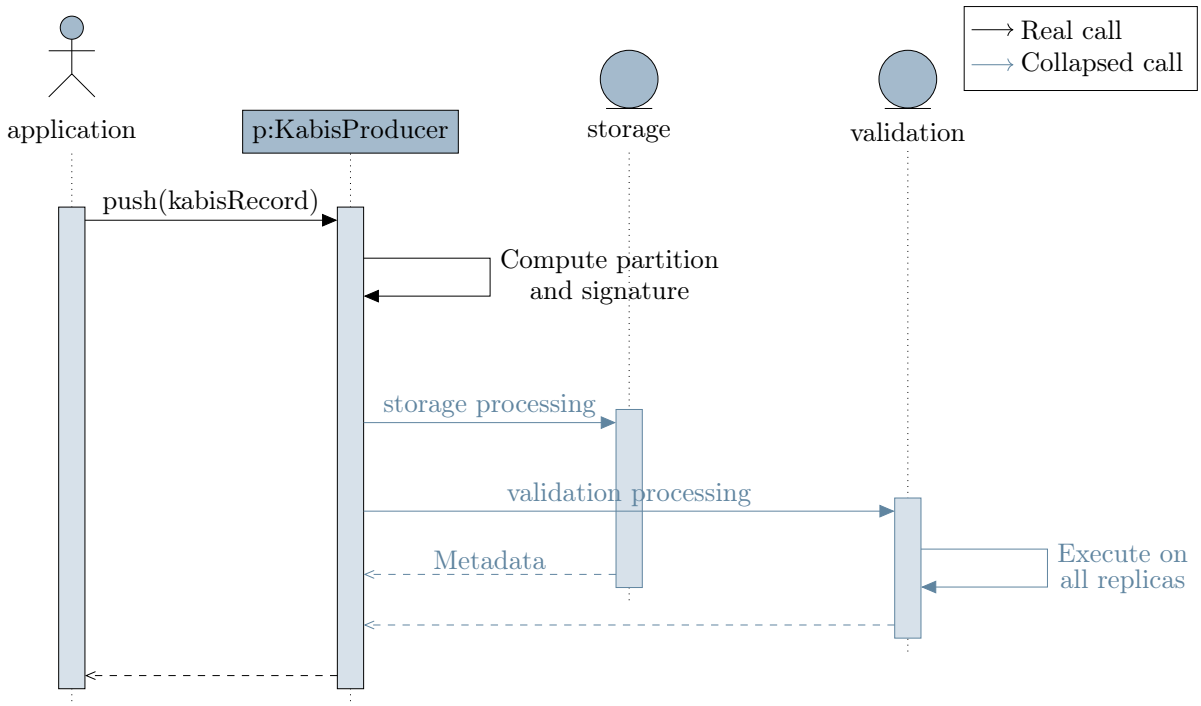

Figure 3.7: Push sequence diagram (validable message)

Storage processing refers to the internal behavior of each `KafkaProducer` to send the given `MessageWrapper` to the respective *Kafka replica*.

First, the `KafkaProducer` wraps the `MessageWrapper` into a `ProducerRecord` as the value. Knowing the topic partition for the event, the producer issues a *metadata request* to the local broker. The response then allows the producer to know the leader for the topic-partition, which may be the local broker or one hosted by another party. The producer then issues the write to the leader, which propagates it to the other replicas.

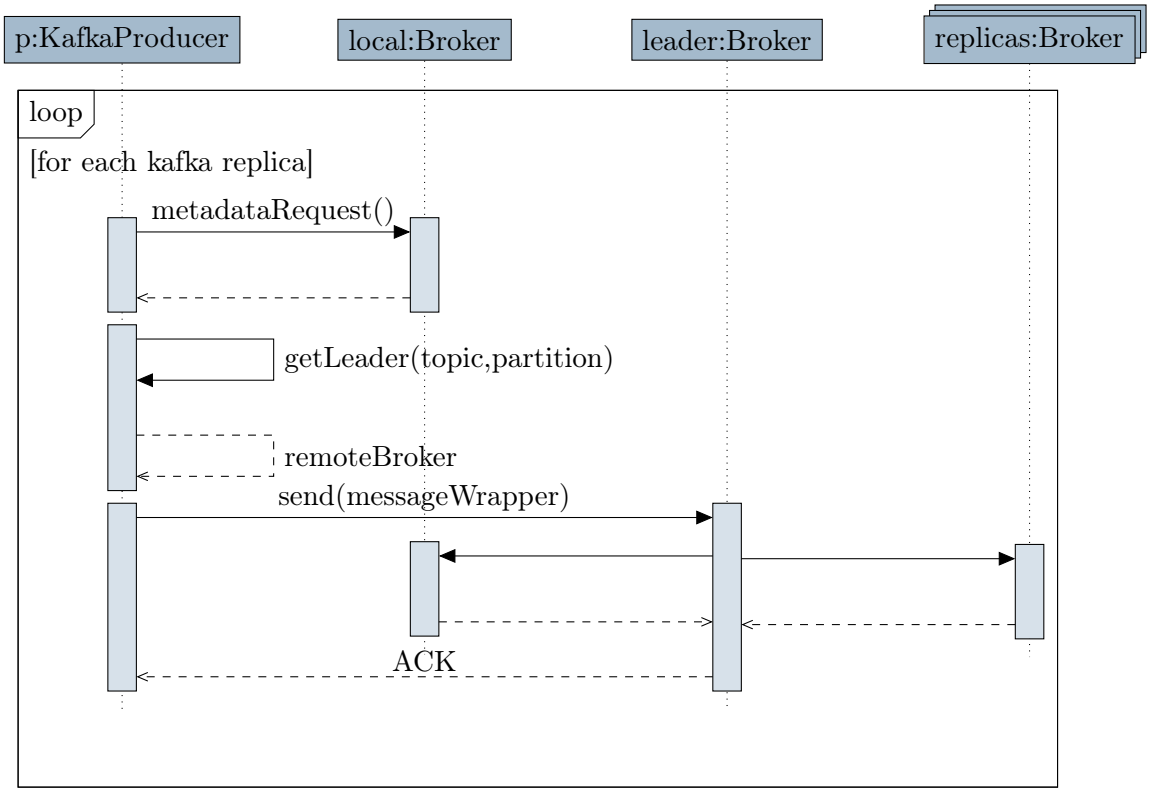

Figure 3.8: Storage push sequence diagram (validable message)

Validation processing abstracts the interaction with the BFT-SMaRt library. The *service proxy* owned by the `KabisProducer` invokes the `<PUSH,SID>` command (serialized as a `byte[]`) on the local *service replica*, which then exploits the MOD-SMaRt protocol [53] (which description can be found in the referenced paper) to atomically and reliably propagate the execution to all the other *service replicas*.

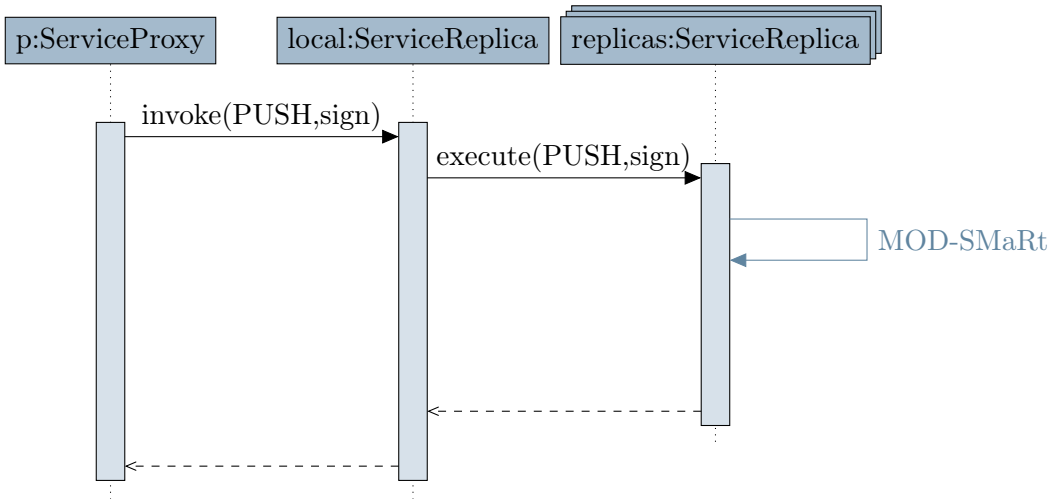

Figure 3.9: Validation push sequence diagram (validable message)

Byzantine fault tolerance is preserved by this protocol: assuming a maximum of F_B byzantine faults, then the *storage layer* is composed of at least $F_B + 1$ *Kafka replicas* and the *validation channel* can operate correctly in presence if this number of faults. Having $F_B + 1$ *Kafka replicas* ensures that at least one of them will be correct.

FIFO order on the validation channel is guaranteed by the sequential behavior of the `KabisProducer`, which will only send a new record after the last one has been correctly stored by the *validation channel* and a sufficient number of *Kafka replicas*.

Total order on the validation channel is guaranteed by the *validation channel* itself, that allows to execute requests preserving total order. The implementation used in Kabis, reported in appendix A, only accepts `PUSH` commands as ordered requests.

The *validable* push protocol therefore preserves the ordering properties of the *validation channel*.

Unvalidable events are published by executing only a subset of the already described operations. Specifically, the `KabisProducer` only computes the partition for the event (avoiding to update the sequence number and computing the signature) and sends the resulting `MessageWrapper` (which will have null `signature` and negative `sequenceNumber` values as flags) only through the first *Kafka replica* of the *storage channel*. The sequence of interactions would be the same described in figure 3.8, except the loop is only executed for the first *Kafka replica*. As a consequence, an *unvalidable* push gets the very same guarantees of the underlying Kafka implementation, which composes *basic guarantee set*.

3.3.2. Pull protocol

`KabisConsumers` consume events with the `Iterable<KabisRecord> pull(Duration)` primitive. The procedure for this operation is more complex, as it involves continuous background processing. The processes involved in a pull sequence are now presented. Then, they are put together to describe the full pull operation.

KafkaPollingThread Inside a `KabisConsumer`, a subprocess called `KafkaPollingThread` caches Kafka events. The cache is first indexed for *Kafka replica*. Among a replica, it aggregates events by `TopicPartitions` and `KabisProducers` (identified by their public keys) and collects them in queue. The thread owns a `KafkaConsumer` for each *Kafka replica* of the *storage channel*. Also, for each replica the cache tracks the (presumably) failed `KabisProducers` into a set, which can be updated or read by other components.

Consider the predicate `isFull(replica)`, 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 3.1 `KafkaPollingThread` main loop

```
1:  K ← set of kafka replicas
2:  for all k ∈ K do
3:    if !isFull(k) then
4:      C ← getKafkaConsumer(K)
5:      R ← pull(C)
6:      for all r ∈ R do
7:        F ← getFailedProducers(k)
8:        p ← getKabisProducer(r)
9:        if p ∉ F then
10:         tp ← getTopicPartition(r)
11:         q ← getQueueFor(k, tp, p)
12:         v ← getValue(r)
13:         push(q, r)
14:       end if
15:     end for
16:   end if
17: end for
```

This code is repeated by the thread indefinitely, until the thread is stopped by calling its `shutdown()` method.

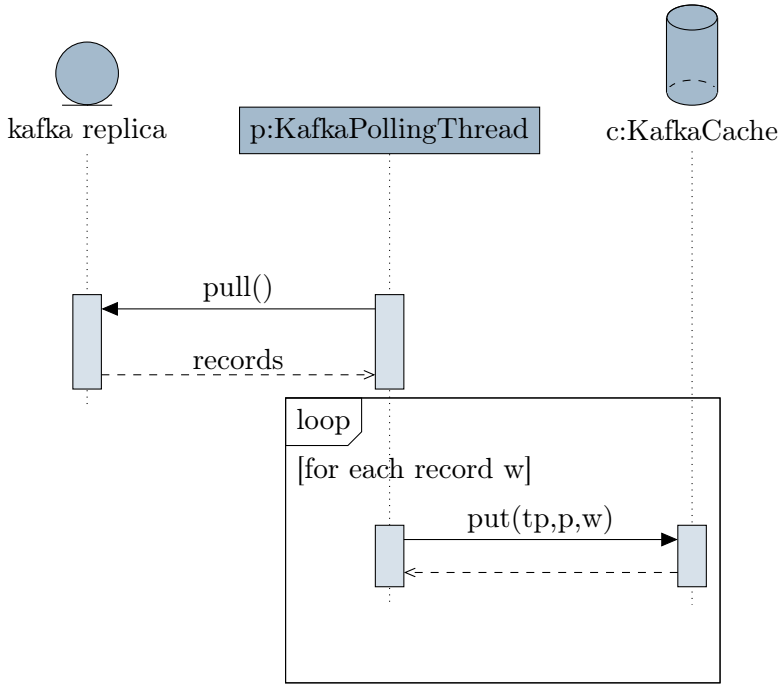

Figure 3.10: KafkaPollingThread loop

Validator The `Validator`'s responsibility is, given a list of SIDs from the *validation channel*, to query all the *Kafka caches* for matching events. This is achieved by the `List<KabisRecord> validate(List<SID>)` function, which pseudo-code is the following:

Algorithm 3.2 `validate` procedure

```
1: S ← list of signatures passed as argument
2: K ← set of kafka replicas
3: R ← ∅
4: for all s ∈ S do
5:    p ← getKabisProducer(s)
6:    tp ← getTopicPartition(s)
7:    for all k ∈ K do
8:       F ← set of failed producers for kafka replica k
9:       if p ∉ F then
10:         result ← null
11:         queue ← getQueueFor(k, tp, p)
12:         if result ≠ null then
13:            wrapper ← poll(queue)
14:            if signatureVerify(s_v) then
15:               result ← w
16:            else
17:               F ← F ∪ {p}
18:            end if
19:         else
20:            pop(queue)
21:         end if
22:         r ← buildKabisRecord(wrapper)
23:         R ← R ∪ {r}
24:      end if
25:   end for
26: end for
```

This code is correct under the following assumption, that will be proven correct in section 3.3.2: "given a correct *Kafka replica* k and a correct `KabisProducer` p, records sent from p will be read from any correct consumer c of k in the same order as they appear in the *validation channel*". Given an SID, if its sender was not marked as failed before for a *Kafka cache*, 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 of the remaining caches (in order to "consume" the replicas of correct event),

and return the found event.

- **The SID does not match** Either the sender has failed, or the *Kafka replica* has. Instruct the cache to mark the sender as invalid, and try with the next cache[3].

If the sender is considered failed, the cache is skipped.

[3]The goal is to mark the <`KabisProducer`,*Kafka replica*> pair as failed. Marking the `KabisProducer` as failed for a cache is functionally equivalent and allows to instruct the cache, which is a local process and therefore can be trusted by its owner, to discard events from the `KabisProducer` and reduce memory usage.

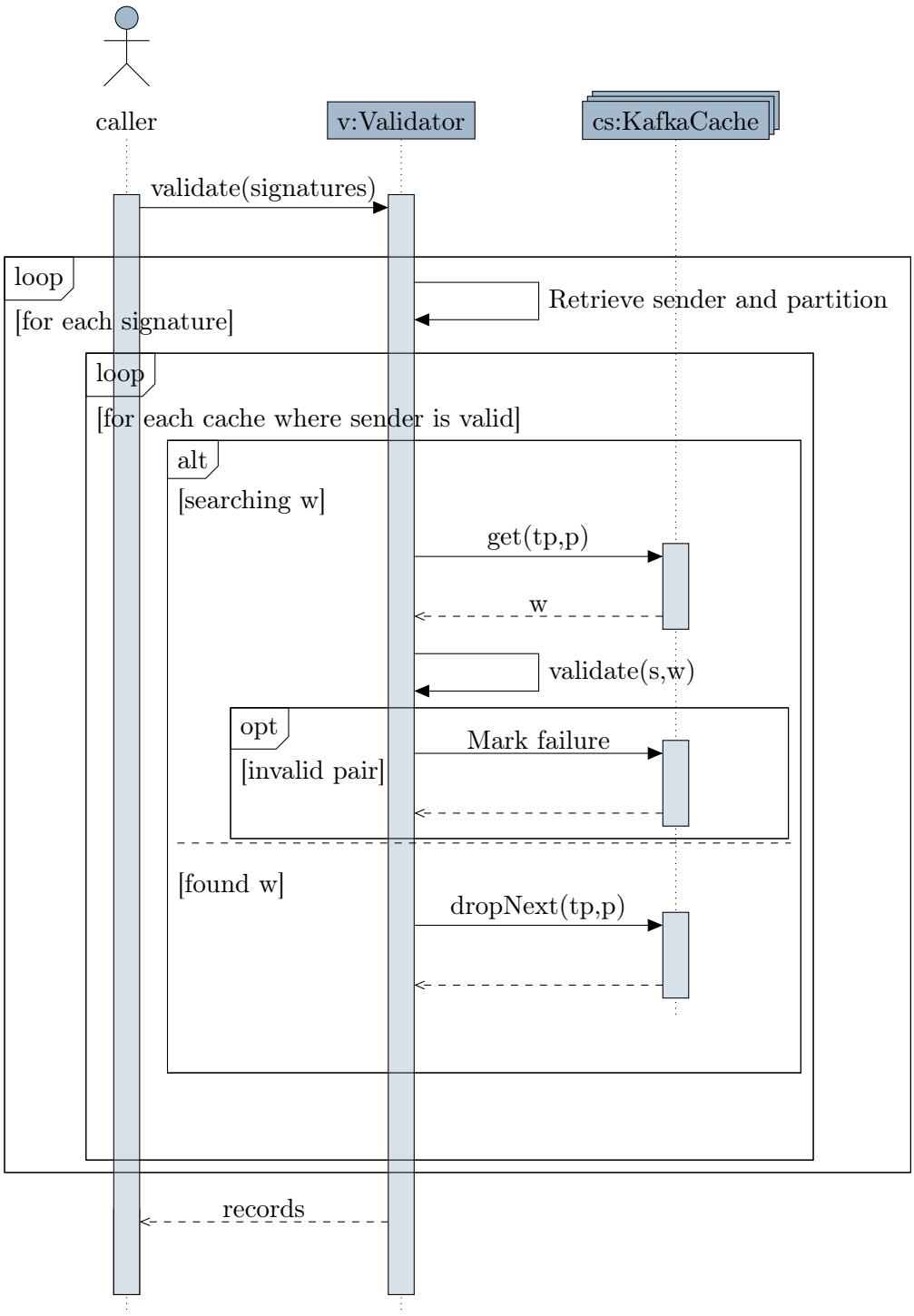

Figure 3.11: Validator procedure

With the `Validator` component, the `pull(Duration)` procedure can finally be implemented. The `KabisConsumer` first pulls from the *validation channel*, getting the new SIDs since the last pull. This is achieved by invoking the `<PULL,offset>` on the *service proxy*. This pull is repeated until the received list of SIDs is not empty, or the given `Duration` expires (in

which case an empty list is returned).

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.

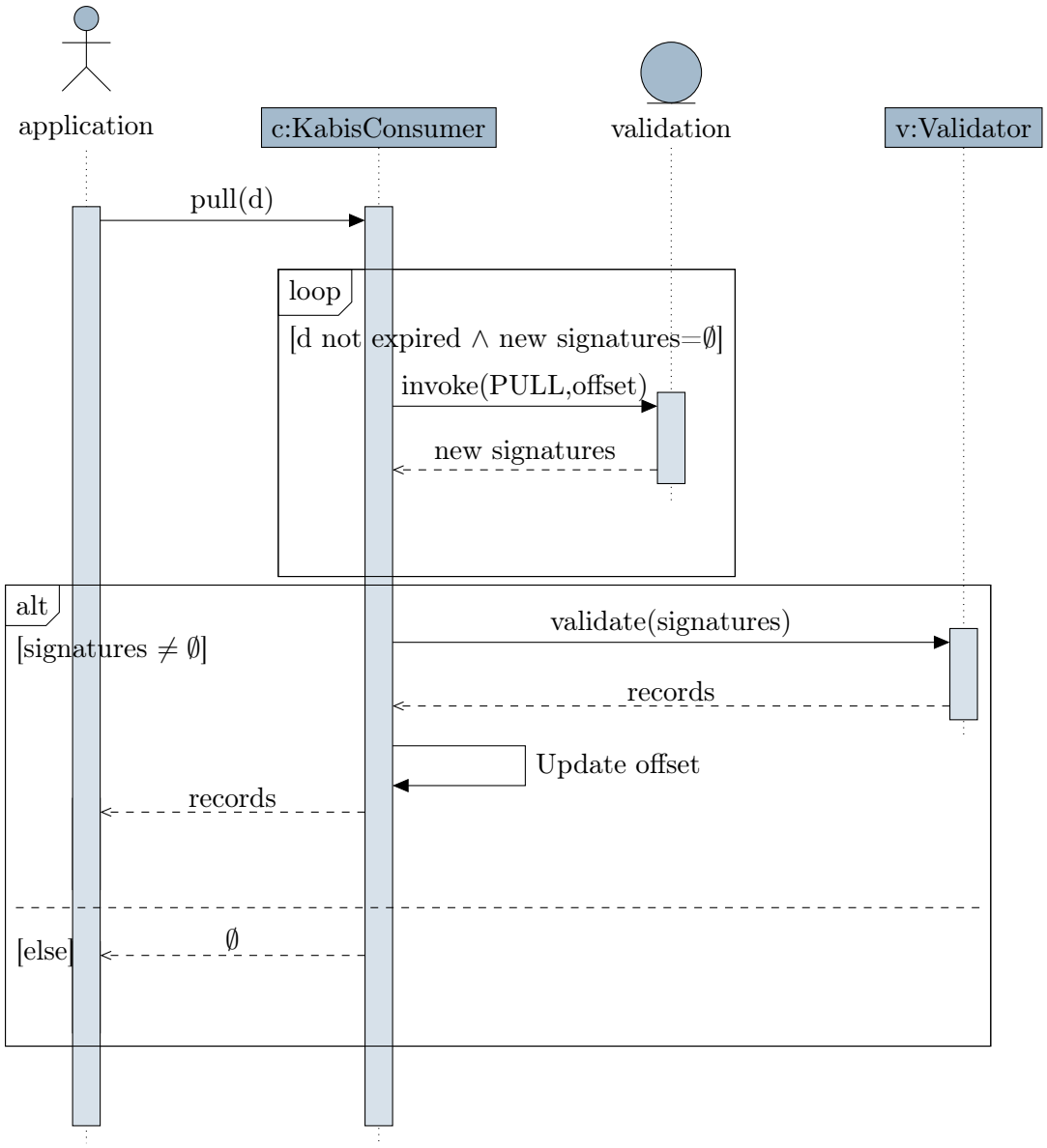

Figure 3.12: Pull sequence diagram

Byzantine fault tolerance As described in section 3.2 the *validation channel* can be deployed so to be resistant to an arbitrary number F_B of byzantine failures. Also, $F_B + 1$ *Kafka replicas* are in the system, and among them at least one is correct for sure. Since the algorithm used for digital signature is assumed to be resistant to preimage attacks,

we can use the signature to rule out any tampered messages from the set of messages read from the *Kafka replicas* and, since at least one is correct for sure, a correct message will always be deliverable. Moreover, because of the `MessageWrapper` implementation and the FIFO ordering guarantee of any (correct) *Kafka replica*, given the next undelivered `SID` we know for sure that the correct `MessageWrapper` will be the first undelivered from the same `KabisProducer` (identified by the `senderId` attribute) from the same `TopicPartition`.

FIFO total order At the end of section 3.3.1 was highlighted how a `KabisProducer` produces events on the *validation channel* in an order that is both FIFO and total. Since a `KabisConsumer` receives a list of `SID` from such channel, and uses it to find matching events in the *storage channel*, the resulting order will be that of the `SID` list, and so that of the *validation channel*.

4 | Performance evaluation

Kabis performance has been empirically evaluated under different configurations. Specifically, the system has been tested varying the number of *validated* topics and the size of the event notifications.

The evaluation metric used is the system throughput, defined as the number of operations the system was able to process per unit of time, and expressed in $Kops/s$. Consumer and producer throughput have been distinguished, because it was suspected that due to the strict decoupling and the higher complexity of the consumer the results would have been different.

The rest of the chapter analyzes the performance of these Kabis when its processes are configured to validate $0, 2, 5, 7$ or 10 topics, out of a total of 10. For both consumers and producers, the measurements found have been first plotted all together to provide a qualitative overview of the magnitudes of the performance changes, and then two specialized plots have been used to distinguish the different behavior when the validation channel was used or not.

4.1. Experiment setup

The experiment setup for Kabis experiments were executed with 1 `KabisConsumer` and 3 `KabisProducers`. The event service was deployed with 2 *Kafka replicas* each internally replicated 4 times (for a total of 8 Zookeeper instances and 8 Kafka brokers), and 4 *service replicas*. This setup guarantees a fault tolerance to a maximum of $F_B = 1$ byzantine failures or $F_C = 3$ crash failures, and is therefore the minimum reasoning setup for a micro-benchmark. Evaluation of Kafka have been performed on an infrastructure equivalent to a single *service replica*, while BFT-SMaRt have been deployed with 4 *service replicas*, modified from Kabis implementation to transmit the event in plain instead of its SID. The processes used for each evaluated system are summarized in table 4.1.

	BFT-SMaRt	Kafka	Kabis
Service replica	4	0	4
Kafka broker	0	4	8
Zookeeper instance	0	4	8
Consumer	1	1	1
Producer	3	3	3
Total	8	12	24

Table 4.1: Processes for each evaluated system

Each process have been inserted into a Docker container, and the experiments ran on a single Linux Mint machine equipped with an AMD Ryzen 9 5900x CPU (offering 24 hardware threads) and 64GB of memory.

Analysis of each of the systems under continuous load allowed shown that in our peculiar setup the amount of time and requests needed for each of the systems to reach a stable state was minimal, allowing to take direct measurements of the throughput instead of relying on more sophisticated techniques.

Therefore, for each presented combination of payload and number of validated topics, the setup has been tested by measuring the execution time required to transmit 50000 events from each producer, and waiting for the consumer to consume 150000 events. The measurement happened on each client: for producers, the client first measured the initial time, then sent all the events through the producer, and finally measured the final time, persisting the difference between the final and the initial time on disk. Regarding the consumer, again its client first measured the initial time. Then it continued to pull the consumer and count the received events, until all the events were read. Finally it measured the final time, computed the difference and persisted it on disk. In post-processing, the throughput of a producer was computed from the stored time t_p as $T_p = \frac{50000}{t_p}$, and that of a consumer that measured an execution time of t_c was computed as $T_c = \frac{150000}{t_c}$.

To reduce the variance of the experiments, each configuration has been tested 5 times, and the resulting throughputs have been averaged.

For Kabis and BFT-SMaRt experiments, the client processes were forced to wait $5s$ before starting their real execution[1], to allow the service replicas to become fully operational. Moreover, for Kabis and Kafka, a first round of communication of 1 event per producer

[1]The client only acquired the initial time after the waiting time to ensure a proper initialization of the *service replicas* elapsed, so to not include this delay in the measured execution time.

was sent on each topic before the benchmark began[2], to prime the Kafka infrastructure and avoid to measure the time needed to create and assign the topics and the consumer groups.

4.2. Performance under varying payloads

The performance requirements of Kabis was to keep consistent performance when performing validation independently from the end-to-end transmitted payload, while achieving performance close to those of Kafka when not doing so. The graphs in figures 4.1 and 4.3 summarize the evolution of Kabis consumer and producer throughput with increasing message payload, compared to those of Kafka ans BFT-SMaRt, showing that this goal has actually been achieved. Figures 4.2 and 4.4 focus instead on the cases when a Kabis process is performing validation (comparing the results with BFT-SMaRt) or not (comparing with Kafka) to better highlight each of the performance requirements.

[2]Even in this case, the priming happened before measuring the initial time.

4.2.1. Consumer evaluation

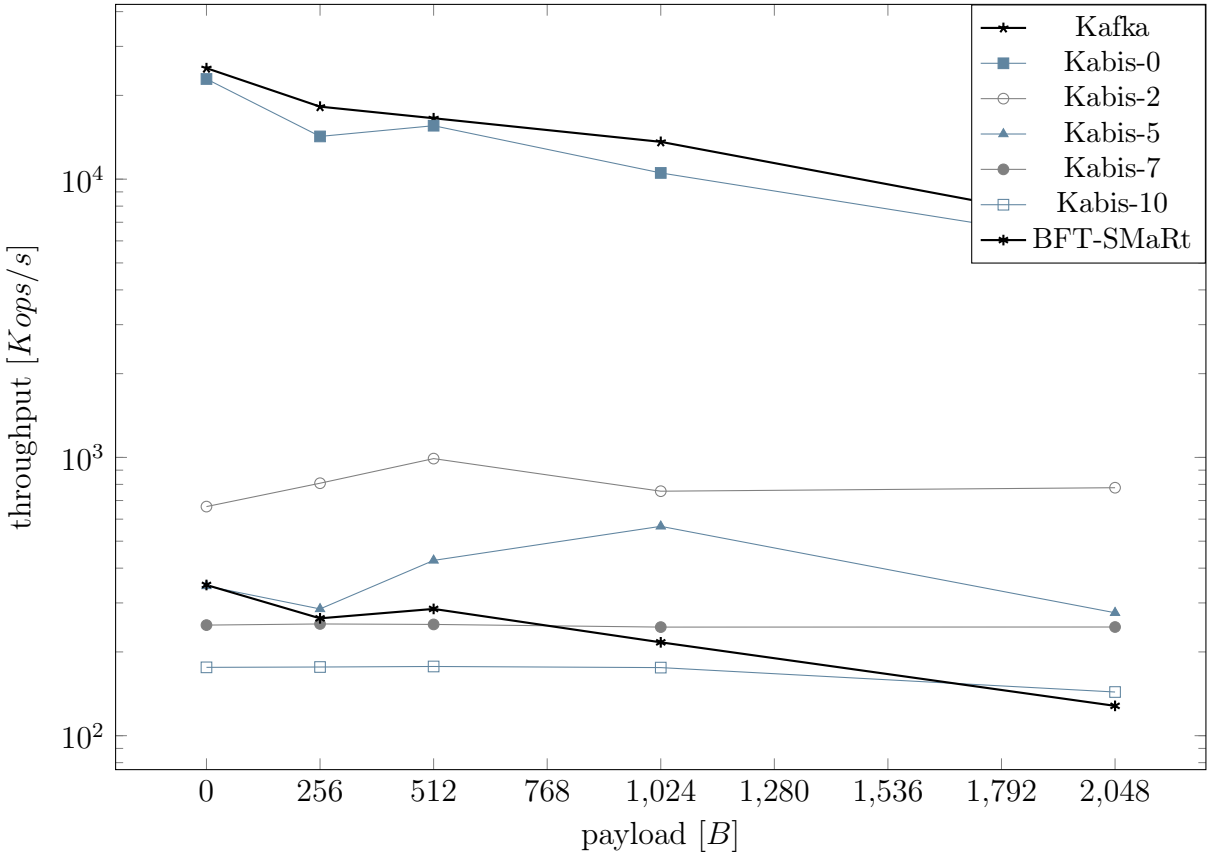

Figure 4.1: Kabis consumer throughput per payload

The graph in figure 4.1 show the evolution of the consumer throughput of Kafka, BFT-SMaRt, and Kabis configured to validate a different number of topics. The y axis is in logarithmic scale to make the curves distinguishable. It can be seen that when validating 0 topics, Kabis performance is close to that of Kafka. As the number of validated topics increases the throughput decreases, and the curve flattens to a constant value as desired.

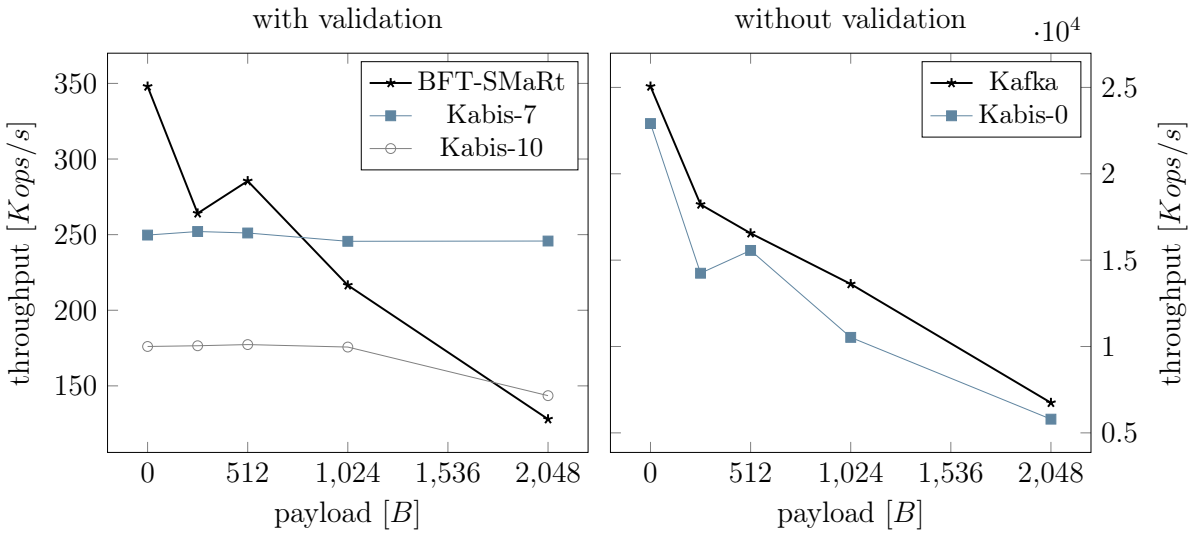

Figure 4.2: Kabis consumer throughput per payload details

Figure 4.2 shows the evolution of the throughput in linear scale for some of the evaluated configurations. The graph on the left highlight the performance loss of BFT-SMaRt as the event payload increases, eventually resulting in Kabis to outperform it. The right graph instead shows how the offset between Kabis consumer throughput and Kafka's appears to be independent from the payload size. It can be said that Kabis consumers meet the performance requirements previously introduced.

4.2.2. Producer evaluation

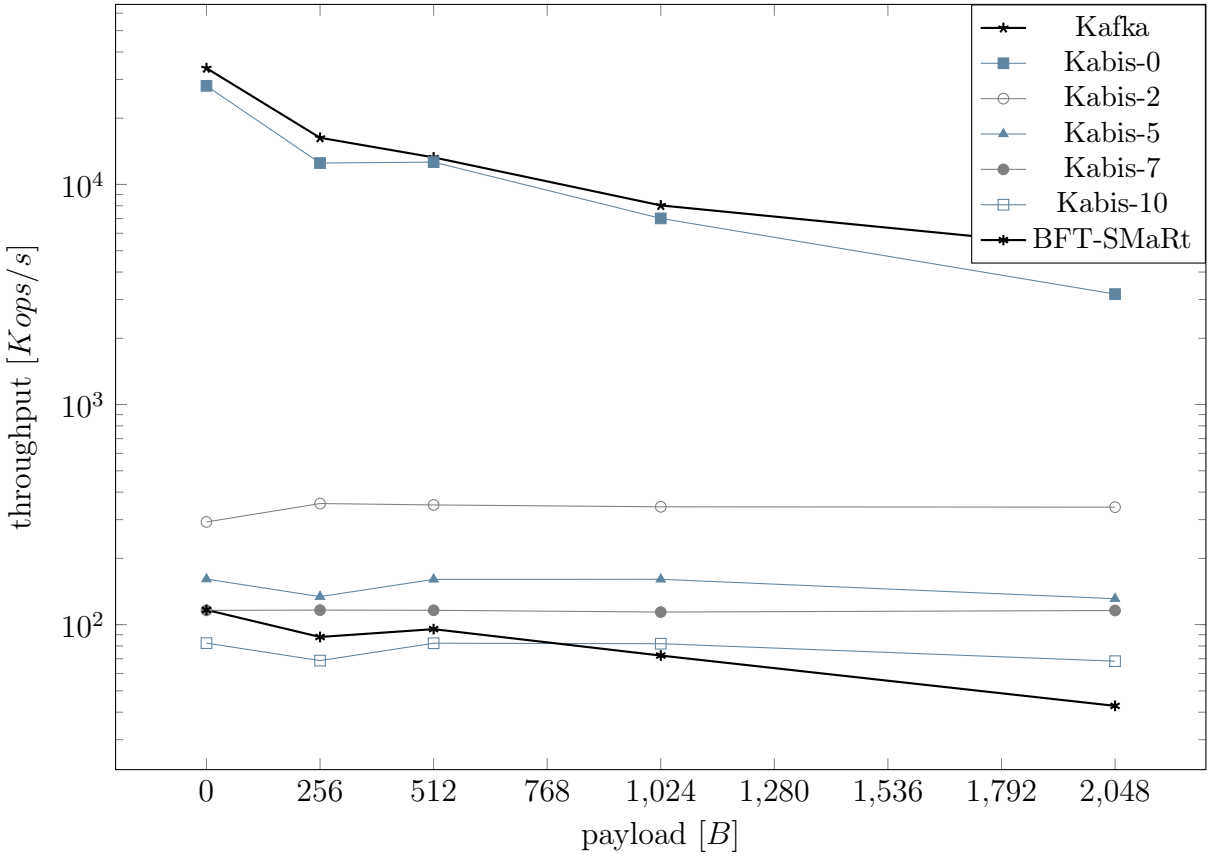

Figure 4.3: Kabis producer throughput per payload

The graph in figure 4.3 plots the evolution of the throughput of the evaluated systems when events of different size are notified. Dually to what happened for the consumers, when a producer is only publishing *unvalidable* events its performance is close to those of Kafka and the throughput decreases as the number of validated topics increases. However even with just 2 validated topics the throughput stabilizes to a constant value, while the consumer's throughput becomes stable only with 7 validated topics.

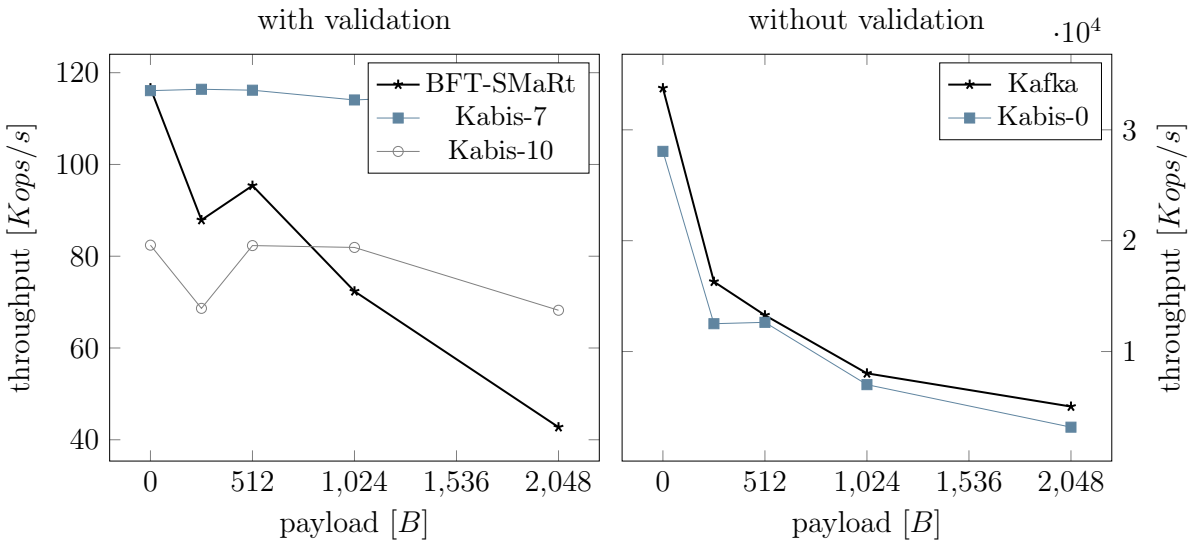

Figure 4.4: Kabis producer throughput per payload details

It can be seen from 4.4 that Kabis producers meet the desired requirements. Specifically, from the left graph it can still be seen that the throughput reduction still does not strictly depends on the payload, and that Kabis eventually outperforms BFT-SMaRt. The graph on the right shows that Kabis producer throughput is not significantly smaller than the one of Kafka.

4.3. Discussion

The graphs in figures 4.1 and 4.3 show a huge performance drop as soon as 2 topics receive validation. Even if this was partially expected, the magnitude of the loss can be surprising. However, it should be considered that the plotted value is obtained by measuring the overall execution time, but no explicit information is available whether the validated and non-validated topics had been produced and consumed at the same rate, or this results mostly reflect the consumer waiting for the last events on some validated topics.

A comparison of the two graphs can however give some insights on this hypothesis. Differently from the consumer, the producer exhibits a constant throughput for any number of validated topics greater than zero. Since the current implementation of the producer is completely sequential, any unvalidable write must wait all of the previous validable writes to complete. This forces the producer to always follow the pace of the validation channel, resulting in lower throughput. Despite not being a real prove, this reasoning allows to notice a possible improvement of the current Kabis design.

4.3.1. Threats to validity

Since the experiments were carried out in a containerized environment hosted on a single machine, the outcome may be affected by the limited resources available to each process and by the absence of network latency. The limitation of the available computational resources however would likely affect Kabis more than the reference systems, since it computes and verifies signatures internally, a relatively costly procedure compared to the standard execution of Kafka and BFT-SMaRt. Regarding the absence of network latency, since BFT-SMaRt's MOD-SMaRt protocol involves an elevated number of message exchange between the service replica, this technology (and Kabis as well, since it relies on it to deliver the `SIDs`) is expected to be the one to benefit more from this advantage.

5 | Related work

In this section, we discuss related work on linking actions and the parties that perform them within a software system.

Event-based communication. Event-based communication has been historically investigated in publish-subscribe systems, which decouple event producers and event consumers in a software architecture [23]. Over the years, software platforms for event-based communications have been increasingly adopted and introduced event processing capabilities to analyze and transform events while they move from producers to consumers [19]. The platform we adopt in this work, Apache Kafka, is becoming the de-facto standard for event-based communication thanks to its scalability and persistency features [54].

Byzantine fault tolerance cost. Different algorithms have been proposed to reduce the required number of replicas for byzantine fault tolerant systems. Veronese et al. [57] developed an algorithm providing byzantine fault tolerance requiring only $2F_B+1$ replicas. This is achieved by trusting a third-party server making it impossible to make conflicting statements without being detected. Kapitza et al. [32] improved over this result by deploying a BFT protocol that only required F_B+1 active replicas during normal operation, and is able to activate F_B extra replicas when fault behavior is suspected.

Accountability. Accountability is about associating states and actions with identities of the parties involved in a system, and provides primitives for actors to validate the states and actions of their peers, such that cheating or misbehavior become detectable, provable, and undeniable by the perpetrator [62]. Jagadeesan et al. [31] study accountability, i.e., after-the-fact verification as a mean to enforce authorization policies. They provide a formal model that accounts for the trade-offs between the power of the auditor, the efficiency of the audit protocol, the requirements on the agents, and the requirements on the communication infrastructure.

Erosion of accountability due to the computerization of society has been discussed for long [46, 47], showing that, without adequate countermeasures, computerization may undermine accountability, reducing the potential for liability to punishments, such as

monetary damages. This is because, for example, software bugs are viewed as inevitable, hence it is often considered unreasonable to hold developers accountable.

Implementing accountability for concrete software systems has led to various approaches. PeerReview [27], is a system that ensures that faults observed by a correct node are properly detected and associated to a faulty node and that a correct node cannot be considered responsible of a fault. This approach has been applied, among others, to a network filesystem, a peer-to-peer system, and an overlay multicast system. Attested Append-Only Memory [14] is a trusted log which ensures that a faulty host cannot lie in different ways to other parties while retaining linearizability and liveness. Dissent [16] is a messaging protocol that offers provable anonymity with accountability preserving message integrity and one-to-one correspondence between members and messages, making denial-of-service attacks by members traceable.

Accountability in cloud computing. At the conceptual level, the TrustCloud framework [35] is a reference model that addresses accountability in cloud computing considering both technical and policy-based aspects.

Concerning system implementations, Zhao et el. [38] propose a scalable, fine-grained access control system based on attribute-based encryption for secure access control in cloud computing. The approach implements user accountability using traitor tracing, i.e., it prevents that dishonest users may share their attribute private keys with other users, who have lower privileges. Accountable Virtual Machines [28] execute software images in a virtualized environment but record non-repudiable information for auditors. This information shows whether the software behaved as intended in distributed systems where hosts do not necessarily trust each other, or third-party platforms for software execution. Finally, Xiao et al [60] provide an accountable MapReduce system for the cloud. Malicious nodes may intentionally corrupt the result of the mapping or reducing phase making the whole final result untrustworthy. In *Accountable MapReduce* a group of auditors checks the workers to detect malicious nodes in real time. A formal model allows one to find the optimal number of workers and auditors to minimize processing time.

Forensic-ready software systems. Forensic readiness is about the availability of the information about how a an incident occurred and who is responsible for it, potentially resulting in a successful prosecution [11, 51]. Researchers have investigated various aspects of forensics readiness, such as ensuring that evidence of potential incidents is preserved [34, 49], that evidence is maintained without alteration [33], as well as software engineering practices for he development of forensic-ready systems [50].

Zawoad et al. [64] propose a *forensics-enabled* mechanism for cloud logs. The goal is to

provide the logs content to investigators still preserving user privacy and logs integrity — in existing logging schemes, the logger is in the trusted computing base hence cloud providers (loggers) can collude with users or investigators to alter the logs. Logs are accessible to investigators only through a RESTful APIs that ensures confidentiality of logs. Similar to our approach, integrity is ensured by hash-chain scheme and proofs of past logs published periodically by the cloud providers.

Similar tot he approach above, *tamper-evident logging* [18] is collected for forensic purposes in the case of an untrusted logger with clients storing their events in the log. A tree-based data structure ensures that the log can efficiently generate proofs to auditors that logged events are still present, and that the current state of the log is consistent with its past content.

Auditability and data verification/possession. Auditability [10] refers to the ability of an auditor to get accurate information when examining a computing system. Poor auditability is associated to a system that has poorly-maintained (or non-existent) records and does not enable efficient auditing of processes. Auditability is an enabler of (retrospective) accountability because an action to be reviewed against a pre-determined policy to decide if the action is compliant, and eventually hold accountable the entity responsible for the action [35]. For example, public auditability for cloud storage correctness [59] ensures that anyone, not just clients who originally stored information on a cloud server, can verify the correctness of the stored data.

Ateniese et al. [5] propose a model for provable data ownership. With this system a client that has stored data in an untrusted server can verify that the server possesses the original data without retrieving it. The underlying mechanism is to generate probabilistic proofs of possession by sampling the blocks on the server. The client verifies the proof. This approach, however does not support dynamic data storage. A follow up work [6] relaxes this constraint but imposes a priori bound on the number of queries and does not support block insertion. Wang et al. [59] provide a mechanism to achieve auditability of cloud storage with data dynamics, i.e., file blocks can be modified, inserted, and deleted. Oruta [58] provides a similar mechanism but the identity of the signer of each block in shared data is kept private from a third party auditor, that publicly verifies the integrity of shared data. More recently, the research in this area has focused on simplifying the implementation and improving performance, e.g., using a skip list for membership queries instead of Merkle trees or 2–3 trees [21, 22].

Non-repudiation. Non-repudiation protocols avoid that a party involved in a a communication can falsely denying taking part in it. For example, the sender of an email

cannot deny sending a message [39]. In blockchain, non-repudiation involves that (1) the sent information cannot be denied and that (2) the information receiver cannot be denied. Digital signatures [45] use asymmetric encryption to guarantee the non-repudiation of information. For example, Xu et al. [61] propose a blockchain-based non-repudiation service provisioning scheme for IoT. In this approach, the blockchain has the role of service publisher and evidence recorder. The verification of a service is based on hashing and validates the (off-chain) service against the on-chain evidence. To this end, the data field of the blockchain transaction also contains the hash value. Similarly, in this work, we use digital signature to ensure that events issued by each party can be unequivocally associated to the sender.

6 | Conclusions

Existing technologies for event-based architectures don't work well 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 as an event system capable of correctly operate in a byzantine, permissioned environment: offering an API similar to that of Apache Kafka, it can be dynamically 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 according to its individual business requirements.

Kabis correctness in presence of byzantine failures has been proved theoretically. 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.

6.1. Future Work

Most of the future development of Kabis is of optimization nature. As already suggested in section 4.2, a stronger decoupling between the storage and the validation channel could probably impact positively the system's performance. Other possible optimizations are the implementation of a sharding mechanism allowing to seek a distinct consensus for each topic or topic-partition, potentially reducing the number of involved processes and therefore the cost of the operation.

Extending `KabisProducer` and `KabisConsumer` API to cover the missing methods from Kafka APIs would be useful to further ease an hypothetical transition from a Kafka-based system to a Kabis-based one. Coordinating BFT-SMaRt state transfer protocol and Kafka crash recovery logic would be another step to make Kabis more of a complete project by adding fault recovery to its capabilities.

Last but not least, despite being build on top of Kafka and exposing a very similar API, Kabis current implementation does not take advantage of Kafka's extraordinary ability to freely scale thanks to consumer groups. Inheriting this feature would greatly increase the complexity of synchronization between the validation channel and the storage channel, but it is probably possible to achieve.

Bibliography

[1] Apache kafka® performance, . URL https://developer.confluent.io/learn/kafka-performance/.

[2] Apache kafka, . URL https://kafka.apache.org/.

[3] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. NDSS'18, 01 2018. doi: 10.14722/ndss.2018.23244.

[4] Amazon AWS. Event Sourcing, 2020. URL https://docs.aws.amazon.com/whitepapers/latest/modern-application-development-on-aws/event-sourcing.html.

[5] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 598–609, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937032. doi: 10.1145/1315245.1315318. URL https://doi.org/10.1145/1315245.1315318.

[6] G. Ateniese, R. Di Pietro, L. V. Mancini, and G. Tsudik. Scalable and efficient provable data possession. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605582412. doi: 10.1145/1460877.1460889. URL https://doi.org/10.1145/1460877.1460889.

[7] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the {Efficiency} of durable state machine replication. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 169–180, 2013.

[8] A. Bessani, J. Sousa, and E. E. Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[9] B. Carminati, C. Rondanini, and E. Ferrari. Confidential business process execution

on blockchain. In *2018 IEEE International Conference on Web Services (ICWS)*, ICWS '18, pages 58–65. IEEE, 2018.

[10] D. Catteddu. Cloud computing: Benefits, risks and recommendations for information security. In C. Serrão, V. Aguilera Díaz, and F. Cerullo, editors, *Web Application Security*, pages 17–17, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16120-9.

[11] S. C. Chan, A. Gehani, J. Cheney, R. Sohan, and H. Irshad. Expressiveness benchmarking for system-level provenance. In *Proceedings of the 9th USENIX Conference on Theory and Practice of Provenance*, TaPP'17, page 3, USA, 2017. USENIX Association.

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

[13] P. Charles. Bessani, alysson and sousa, joão and alchieri, eduardo ep, 2022. URL `https://github.com/bft-smart/library`.

[14] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 189–204, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935915. doi: 10.1145/1294261.1294280. URL `https://doi.org/10.1145/1294261.1294280`.

[15] M. Correia, N. F. Neves, and P. Veríssimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *The Computer Journal*, 49(1):82–96, 2006.

[16] H. Corrigan-Gibbs and B. Ford. Dissent: Accountable anonymous group messaging. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 340–350, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450302456. doi: 10.1145/1866307.1866346. URL `https://doi.org/10.1145/1866307.1866346`.

[17] K. Croman, C. Decker, I. Eyal, A. E. Gencer, A. Juels, A. Kosba, A. Miller, P. Saxena, E. Shi, E. Gün Sirer, D. Song, and R. Wattenhofer. On scaling decentralized blockchains. In J. Clark, S. Meiklejohn, P. Y. Ryan, D. Wallach, M. Brenner, and K. Rohloff, editors, *Financial Cryptography and Data Security*, pages 106–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53357-4.

[18] S. A. Crosby and D. S. Wallach. Efficient data structures for tamper-evident logging. In *Proceedings of the 18th Conference on USENIX Security Symposium*, SSYM'09, page 317–334, USA, 2009. USENIX Association.

[19] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3), June 2012. ISSN 0360-0300. doi: 10.1145/2187671.2187677. URL https://doi.org/10.1145/2187671.2187677.

[20] M. El-Hindi, M. Heyden, C. Binnig, R. Ramamurthy, A. Arasu, and D. Kossmann. Blockchaindb - towards a shared database on blockchains. In *Procs. of the Int. Conf. on Management of Data*, SIGMOD '19, page 1905–1908, New York, NY, USA, 2019. ACM.

[21] C. C. Erway, A. Küpçü, C. Papamanthou, and R. Tamassia. Dynamic provable data possession. 17(4), Apr. 2015. ISSN 1094-9224. doi: 10.1145/2699909. URL https://doi.org/10.1145/2699909.

[22] E. Esiner, A. Kachkeev, S. Braunfeld, A. Küpçü, and O. Özkasap. Flexdpdp: Flexlist-based optimized dynamic provable data possession. *ACM Trans. Storage*, 12(4), Aug. 2016. ISSN 1553-3077. doi: 10.1145/2943783. URL https://doi.org/10.1145/2943783.

[23] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857078. URL https://doi.org/10.1145/857076.857078.

[24] M. Fowler. Event sourcing, 2005. URL https://martinfowler.com/eaaDev/EventSourcing.html.

[25] Google. Deploying event-sourced systems with cloud spanner, 2020. URL https://cloud.google.com/solutions/deploying-event-sourced-systems-with-cloud-spanner.

[26] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[27] A. Haeberlen, P. Kouznetsov, and P. Druschel. Peerreview: Practical accountability for distributed systems. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 175–188, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595935915. doi: 10.1145/1294261.1294279. URL https://doi.org/10.1145/1294261.1294279.

[28] A. Haeberlen, P. Aditya, R. Rodrigues, and P. Druschel. Accountable virtual machines. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 119–134, USA, 2010. USENIX Association.

[29] H. Howard and R. Mortier. Paxos vs raft: Have we reached consensus on distributed consensus? In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–9, 2020.

[30] R. Hull. Blockchain: Distributed event-based processing in a data-centric world. In *Procs. of the Int. Conf. on Distributed and Event-Based Systems*, DEBS '17, page 2–4, New York, NY, USA, 2017. ACM.

[31] R. Jagadeesan, A. Jeffrey, C. Pitcher, and J. Riely. Towards a theory of accountability and audit. In M. Backes and P. Ning, editors, *Computer Security – ESORICS 2009*, pages 152–167, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04444-1.

[32] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 295–308, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312233. doi: 10.1145/2168836.2168866. URL `https://doi.org/10.1145/2168836.2168866`.

[33] V. R. Kebande and H. Venter. On digital forensic readiness in the cloud using a distributed agent-based solution: issues and challenges. *Australian Journal of Forensic Sciences*, 50(2):209–238, June 2016. doi: 10.1080/00450618.2016.1194473. URL `https://doi.org/10.1080/00450618.2016.1194473`.

[34] J. King, J. Stallings, M. Riaz, and L. Williams. To log, or not to log: using heuristics to identify mandatory log events – a controlled experiment. *Empirical Software Engineering*, 22(5):2684–2717, Aug. 2016. doi: 10.1007/s10664-016-9449-1. URL `https://doi.org/10.1007/s10664-016-9449-1`.

[35] R. K. L. Ko, P. Jagadpramana, M. Mowbray, S. Pearson, M. Kirchberg, Q. Liang, and B. S. Lee. Trustcloud: A framework for accountability and trust in cloud computing. In *2011 IEEE World Congress on Services*, pages 584–588, 2011.

[36] L. Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[37] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem.

ACM Transactions on Programming Languages and Systems, pages 382–401, July 1982. URL https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/.

[38] J. Li, G. Zhao, X. Chen, D. Xie, C. Rong, W. Li, L. Tang, and Y. Tang. Fine-grained data access control systems with user accountability in cloud computing. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 89–96, 2010.

[39] P. Louridas. Some guidelines for non-repudiation protocols. *SIGCOMM Comput. Commun. Rev.*, 30(5):29–38, Oct. 2000. ISSN 0146-4833. doi: 10.1145/505672.505676. URL https://doi.org/10.1145/505672.505676.

[40] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 17–30, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450341394. doi: 10.1145/2976749.2978389. URL https://doi.org/10.1145/2976749.2978389.

[41] N. A. Lynch. *Distributed algorithms*. Elsevier, 1996.

[42] T. McConaghy, R. Marques, A. Müller, D. De Jonghe, T. McConaghy, G. McMullen, R. Henderson, S. Bellemare, and A. Granzotto. Bigchaindb: A scalable blockchain database, 2016.

[43] Microsoft. Command and query responsibility segregation (cqrs) pattern, 2020. URL https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs.

[44] Z. Milosevic, M. Hutle, and A. Schiper. On the reduction of atomic broadcast to consensus with byzantine faults. In *2011 IEEE 30th International Symposium on Reliable Distributed Systems*, pages 235–244. IEEE, 2011.

[45] National Institute of Standards and Technology. Digital signature standard (dss). FIPS Publication 186, May .

[46] H. Nissenbaum. Computing and accountability. *Commun. ACM*, 37(1):72–80, Jan. 1994. ISSN 0001-0782. doi: 10.1145/175222.175228. URL https://doi.org/10.1145/175222.175228.

[47] H. Nissenbaum. Accountability in a computerized society. *Science and Engineering Ethics*, 2(1):25–42, 1996. doi: 10.1007/BF02639315.

[48] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (Usenix ATC 14)*, pages 305–319, 2014.

[49] L. Pasquale, S. Hanvey, M. Mcgloin, and B. Nuseibeh. Adaptive evidence collection in the cloud using attack scenarios. *Comput. Secur.*, 59(C):236–254, June 2016. ISSN 0167-4048. doi: 10.1016/j.cose.2016.03.001. URL `https://doi.org/10.1016/j.cose.2016.03.001`.

[50] L. Pasquale, D. Alrajeh, C. Peersman, T. Tun, B. Nuseibeh, and A. Rashid. Towards forensic-ready software systems. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*, pages 9–12, 2018.

[51] R. Rowlingson. A ten step process for forensic readiness. *International Journal of Digital Evidence, IJDE*, 2, 01 2004.

[52] H. Shafagh, L. Burkhalter, A. Hithnawi, and S. Duquennoy. Towards blockchain-based auditable storage and sharing of iot data. In *Procs. of the Cloud Computing Security Workshop*, CCSW '17, page 45–50, New York, NY, USA, 2017. ACM.

[53] J. Sousa and A. 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.

[54] B. Stopford. *Designing Event-Driven Systems: Concepts and Patterns for Streaming Services with Apache Kafka*. O'Reilly Media, 1st edition, 2018. ISBN 9781492038245.

[55] T. Sund, C. Loof, S. Nadjm-Tehrani, and M. Asplund. Blockchain-based event processing in supply chains - a case study at ikea. *Robotics and Computer-Integrated Manufacturing*, 65:1–16, 2020.

[56] M. Swanson, J. Hash, and P. Bowen. Sp 800-18 rev. 1. guide for developing security plans for federal information systems, 2006.

[57] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2013. doi: 10.1109/TC.2011.221.

[58] B. Wang, B. Li, and H. Li. Oruta: privacy-preserving public auditing for shared data in the cloud. *IEEE Transactions on Cloud Computing*, 2(1):43–56, 2014. doi: 10.1109/TCC.2014.2299807.

[59] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):847–859, 2011. doi: 10.1109/TPDS.2010.183.

[60] Z. Xiao and Y. Xiao. Achieving accountable mapreduce in cloud computing. *Future Generation Computer Systems*, 30:1 – 13, 2014. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2013.07.001. URL `http://www.sciencedirect.com/science/article/pii/S0167739X13001465`. Special Issue on Extreme Scale Parallel Architectures and Systems, Cryptography in Cloud Computing and Recent Advances in Parallel and Distributed Systems, ICPADS 2012 Selected Papers.

[61] Y. Xu, J. Ren, G. Wang, C. Zhang, J. Yang, and Y. Zhang. A blockchain-based non-repudiation network computing service scheme for industrial iot. *IEEE Transactions on Industrial Informatics*, 15(6):3632–3641, 2019. doi: 10.1109/TII.2019.2897133.

[62] A. R. Yumerefendi and J. S. Chase. The role of accountability in dependable distributed systems. In *Proceedings of the First Conference on Hot Topics in System Dependability*, HotDep'05, page 3, USA, 2005. USENIX Association.

[63] M. Zamani, M. Movahedi, and M. Raykova. RapidChain: scaling blockchain via full sharding. In *Procs. of the Conf. on Computer and Communications Security*, CCS '18, page 931–948, New York, NY, USA, 2018. ACM.

[64] S. Zawoad, A. K. Dutta, and R. Hasan. Towards building forensics enabled cloud through secure logging-as-a-service. *IEEE Transactions on Dependable and Secure Computing*, 13(2):148–162, 2016. doi: 10.1109/TDSC.2015.2482484.

A | Validation service replica implementation

```
1  package kabis.validation;
2
3  import bftsmart.tom.MessageContext;
4  import bftsmart.tom.server.defaultservices.DefaultSingleRecoverable;
5  import org.apache.kafka.common.errors.SerializationException;
6  import org.bouncycastle.jce.provider.BouncyCastleProvider;
7  import org.slf4j.Logger;
8  import org.slf4j.LoggerFactory;
9
10 import java.io.*;
11 import java.nio.ByteBuffer;
12 import java.security.Security;
13 import java.util.ArrayList;
14 import java.util.LinkedList;
15 import java.util.List;
16
17 import static kabis.validation.KabisServiceReplica.deserializeSidList;
18 import static kabis.validation.KabisServiceReplica.serializeSidList;
19
20 public class KabisServiceReplica extends DefaultSingleRecoverable {
21
22     private static final Logger LOG = LoggerFactory.getLogger(
    KabisServiceReplica.class);
23     private final List<SecureIdentifier> log = new LinkedList<>();
24
25     @Override
26     public void installSnapshot(byte[] bytes) {
27         log.clear();
28         log.addAll(deserializeSidList(bytes));
29     }
30
31     @Override
32     public byte[] getSnapshot() {
```

```
33          return pull(0);
34      }
35
36      @Override
37      public byte[] appExecuteOrdered(byte[] bytes, MessageContext
    messageContext) {
38          try (var cmd = new ByteArrayInputStream(bytes)){
39              var opOrdinal = cmd.read();
40              var op = OPS.values()[opOrdinal];
41              switch (op){
42                  case PUSH:
43                      push(cmd.readAllBytes());
44                      return new byte[0];
45                  case PULL:
46                      var index = ByteBuffer.wrap(cmd.readNBytes(Integer.
    BYTES)).getInt();
47                      return pull(index);
48                  default:
49                      throw new IllegalArgumentException(String.format("
    Illegal ordered operation requested: %s",op));
50              }
51          } catch (IOException e) {
52              throw new SerializationException(e);
53          }
54      }
55
56      @Override
57      public byte[] appExecuteUnordered(byte[] bytes, MessageContext
    messageContext) {
58          try (var cmd = new ByteArrayInputStream(bytes)){
59              var opOrdinal = cmd.read();
60              if (opOrdinal == OPS.PULL.ordinal()) {
61                  var index = ByteBuffer.wrap(cmd.readNBytes(Integer.BYTES
    )).getInt();
62                  return pull(index);
63              }
64              throw new IllegalArgumentException(String.format("Illegal
    ordered operation requested: %s", OPS.values()[opOrdinal]));
65          } catch (IOException e) {
66              throw new SerializationException(e);
67          }
68      }
69
70      public KabisServiceReplica(int id){
```

```
71          new bftsmart.tom.ServiceReplica(id,this,this);
72      }
73
74      public static void main(String[] args) {
75          if (args.length != 1) {
76              LOG.error("USAGE: {} <process id>", KabisServiceReplica.
    class.getCanonicalName());
77              System.exit(-1);
78          }
79          Security.addProvider(new BouncyCastleProvider());
80          int processId = Integer.parseInt(args[0]);
81          new KabisServiceReplica(processId);
82      }
83
84      private void push(byte[] serializedSid){
85          var sid = SecureIdentifier.deserialize(serializedSid);
86          synchronized (log) {
87              log.add(sid);
88          }
89      }
90
91      private byte[] pull(int index){
92          if(index>log.size()) return new byte[0];
93          List<SecureIdentifier> logPortion;
94          synchronized (log) {
95              logPortion = new ArrayList<>(log.subList(index, log.size()))
    ;
96          }
97          return serializeSidList(logPortion);
98      }
99
100     public static byte[] serializeSidList(List<SecureIdentifier> subLog)
    {
101         try(var bytes = new ByteArrayOutputStream()) {
102             for (var sid: subLog){
103                 var serialized = sid.serialize();
104                 bytes.writeBytes(ByteBuffer.allocate(Integer.BYTES).
    putInt(serialized.length).array());
105                 bytes.writeBytes(serialized);
106             }
107             return bytes.toByteArray();
108         } catch (IOException e) {
109             throw new RuntimeException(e);
110         }
```

```
111     }
112
113     public static List<SecureIdentifier> deserializeSidList(byte[]
    serialized){
114         try (var bytes = new ByteArrayInputStream(serialized)){
115             List<SecureIdentifier> res = new LinkedList<>();
116             while (bytes.available()>0){
117                 var len = ByteBuffer.wrap(bytes.readNBytes(Integer.BYTES
    )).getInt();
118                 var serializedSid = bytes.readNBytes(len);
119                 res.add(SecureIdentifier.deserialize(serializedSid));
120             }
121             return res;
122         } catch (IOException e) {
123             throw new SerializationException(e);
124         }
125     }
126
127 }
```

List of Figures

List of Tables

List of Definitions

Term	Definition	Page
Event	A significant change in a process' state	1
Event notification	The operation by which any process is made aware of an event happened in the context of any other process	1
Event handling	The reaction of a process to an event	1
Event-based system	A system where processes communicate between each other through event notification, and their execution is limited to event handling.	1
Event-sourcing system	An event-based system in which the history of events is stored into a persistent data store, called the event log	1
Producer	A client of an event-based system publishing events	3
Consumer	A client of an event-based system subscribing for events and receiving event notifications	3
Event service	The intermediate service coordinating an event-based system	3
Topic	A semantic class for events of the same type that may happen in an event-based system	4
FIFO order	Messages from a process will be delivered in the same order they were sent	6
Total order	For any couple of messages, one will always be delivered before the other.	6
Crash failure	A failure model in which a process stops responding to incoming messages	5
Byzantine failure	A failure model in which a process exhibits arbitrary, possibly malicious, behavior	5
Crash fault tolerance (CFT)	The ability of a system to correctly operate in presence of some crash failures	5

Term	Definition	Page
Byzantine fault tolerance (BFT)	The ability of a system to correctly operate in presence of some byzantine failures	5
Consensus	When of replicas reach an agreement over some value	6
Permissioned environment	An execution environment where exists a way to determine the identity of any participant	7
Non-repudiation	Assurance that neither the sender or the receiver can later deny having processed the information	7
Authenticity	Allows the receiver of a message to identify the sender	7
Integrity	Allows the receiver to verify that the content of a message has not been modified	7
Digital signature	Technique ensuring authenticity and integrity of a message, therefore achieving non-repudiation	7
SHA-256	A common hashing algorithm	7
ECDSA	An asymmetric encryption algorithm	7
Apache Kafka	Reference event-based technology for this thesis	7
KafkaProducer	Client process that writes events to Kafka	8
KafkaConsumer	Client process that reads events from Kafka	8
Topic	Abstraction used by Kafka to group events of the same type	8
Partition	Subset of a topic created for scalability reasons	8
Offset	Tracks the position where a Consumer is reading	8
Broker	Server process that stores some of the topic-partition of the Kafka system	8
Zookeeper	Technology used by Kafka for broker coordination	8
BFT-SMaRt	An open-source framework providing BFT consensus	9
VP-Consensus	A stronger consensus primitive enforcing the satisfaction of a predicate from any agreed value and the production of a cryptographic proof certifying the agreement upon a specific value in a specific consensus instance	9
Service replica	Service process of BFT-SMaRt.	9
Service proxy	Client process of BFT-SMaRt.	9
Basic guarantees	The set of guarantees offered by Kabis when the receiver doesn't try to validate incoming messages	12

Term	Definition	Page
Extended guarantees	The set of guarantees achieved when both the sender and the receiver aim to get them	13
KabisRecord	Data object used into the messaging layer	14
KabisProducer	API used to send messages to the messaging layer	14
KabisConsumer	API used to read messages from the messaging layer	14
Storage channel	Communication channel internal to the messaging layer offering the basic guarantee set	19
Validation channel	Communication channel into the messaging layer offering extended guarantees	19
MessageWrapper	Data type transmitted through the storage channel	20
Kafka replica	A fully working kafka infrastructure composing by the storage channel	20
Secure Identifier (SID)	Data type transmitted through the validation channel	21

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. Professor Guido Salvaneschi and Leon Chemnitz have my gratitude as well for their 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.