# SmartStream: Towards Byzantine Resilient Data Streaming

Johannes Köstler
Hans P. Reiser
jk@sec.uni-passau.de
hr@sec.uni-passau.de
University of Passau

Gerhard Habiger
Franz J. Hauck
gerhard.habiger@uni-ulm.de
franz.hauck@uni-ulm.de
Ulm University

## ABSTRACT

Data streaming platforms connect heterogeneous services through the publish-subscribe paradigm. Currently available platforms provide protection against crash faults, but are not resistant against Byzantine faults like arbitrary hardware faults and intrusions. State machine replication can provide this protection, but the higher resource requirements and the more elaborated communication primitives usually result in a higher overall complexity and a non-negligible performance degradation. This is especially true for data streaming if the default textbook approach of integrating the service into a replicated state machine is followed without further adaptions. The standard state management with state logs and snapshots and without any partitioning scheme limits both performance and scalability in a way those systems become unusable in practice. That is why we propose SmartStream, a topic-based Byzantine fault-tolerant data streaming platform that harmonizes the competing concepts of both systems and leverages the specific characteristics of data streaming, namely the append-only semantics of the application state and its partitionable structure. We show its effectiveness in a prototype implementation and evaluate its performance. The evaluation results show a moderate drop in system throughput when compared to state-of-the-art data streaming platforms like Apache Kafka, but reasonable overall performance considering the stronger resilience guarantees.

## CCS CONCEPTS

• **Computer systems organization → Reliability**; • **Computing methodologies →** *Distributed algorithms*; • **Software and its engineering → Message oriented middleware**;

## KEYWORDS

Byzantine fault tolerance, streaming platform, publish-subscribe, state machine, replication, message broker

**ACM Reference Format:**
Johannes Köstler, Hans P. Reiser, Gerhard Habiger, and Franz J. Hauck. 2021. SmartStream: Towards Byzantine Resilient Data Streaming. In *The 36th ACM/SIGAPP Symposium on Applied Computing (SAC '21), March 22–26, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3412841.3441904

## 1 INTRODUCTION

Stream processing platforms are the backbone of many modern applications. They operate at the heart of message-oriented middleware components as well as enterprise messaging systems, service-oriented architectures, distributed cloud applications and IoT systems. Such platforms utilize the publish-subscribe paradigm to enable loosely coupled communication architectures and to achieve a high degree of scalability. In contrast to classical client-server communication where messages are passed directly, in publish-subscribe communication the messages are mediated through broker components. Independent publishers can give information into the message stream, and independent subscribers can extract information from it for further processing. Since these platforms provide a core functionality for so many systems, a reliable and scalable execution is fundamentally important.

Available data streaming platforms have recently become increasingly popular, e.g., Apache Kafka and Amazon Kinesis. Both use passive replication and sharding to achieve reliability and scalability. Backup nodes can take over for the main node of the same shard. A single node is typically involved in multiple shards in mostly different roles. The system is centrally organized—there is a central point where all subscriptions and assignments are managed. This kind of architecture can usually be implemented at reasonable resource costs and without substantial degradation of performance. At the same time, however, the effectiveness of these measures in terms of dependability falls short of what is possible, as only crash faults can be tolerated and the delays introduced by fault detection and recovery remain high. With the help of active replication, recovery times in the event of faults could be reduced. And, on the basis of a Byzantine fault model, active replication can also provide protection from corrupted messages and data introduced by unreliable hardware components or by attackers controlling a limited number of replicas.

Using a Byzantine fault model seems like a natural step, as data streaming platforms are nowadays integrated into safety-critical systems like supervisory control and data acquisition (SCADA) systems [19], where the propagation of arbitrary faults can lead to devastating effects. Although Byzantine fault models are not yet considered within data centers and business environments, we see a need of their adoption in classical data streaming applications like monitoring and analytics systems, as the processing of incorrect or modified data can have serious consequences as well. Such corruptions may delay necessary actions and decisions, which ultimately will lead to much greater financial losses than a resilient operation would have caused in the first place. The general motivation for a resilient execution of IT systems manifests itself in temporally recurring reports of soft errors or arbitrary faults in data center environments [1, 7].

We are not aware of any production-ready streaming platform that offers protection against Byzantine faults. State-Machine Replication (SMR) middleware has already demonstrated its applicability in providing this safeguard in various services such as distributed file systems [6], dependable tuple spaces [4], as well as replicated databases [12] and directory services [26]. These implementations claim reasonable performance results, even though SMR comes with negative impacts on those system in terms of resource costs and execution performance. This is because active replication in the Byzantine fault model usually requires more replicas than for crash faults and therefore more communication to reach agreement. It also needs deterministic request execution compared to passive replication to maintain data integrity.

A naive application of SMR concepts to data streaming platforms leads to a drastic decay of the overall performance. This is mainly due to the functional mechanics of the high-throughput-oriented and disk-state-based service that interfere with SMR middleware concepts. However, it turned out that these disadvantages can be overcome by optimizations that reflect the service characteristics and integrate the service layer deeper into the fault tolerance layer. These optimizations include (i) an efficient state management which utilizes the append-only semantics of the service state, (ii) a scalable clustering scheme that makes use of the highly partitionable state structure as well as (iii) a harmonized batching strategy and optimized request execution.

In this paper we provide a discussion of the limitations that emerge when SMR is applied to a data streaming service. Based on that, we propose a new approach for a resilient and scalable SMR based data streaming platform that remedies those limitations. The platform is resilient in the sense of its ability to withstand Byzantine faults and scalable in the sense of its ability to efficiently process large amounts of data. We also contribute a prototype implementation called SmartStream which is based on the data streaming platform Apache Kafka and the SMR middleware BFT-SMaRt. And we further demonstrate its efficiency and effectiveness in a detailed evaluation. All these evaluations contain comparisons with the unmodified Apache Kafka streaming platform and show that the throughput experiences a moderate drop while latency rises reasonably compared to the original implementation. However, we consider these degradations to be acceptable for most services, taking into consideration the increased resilience guarantees and scaling possibilities.

The remainder of this paper is structured as follows. The next section discusses the problems that arise when SMR is applied to a data streaming platform. We then present our integrated data streaming platform in Section 3. This is followed by Section 4 covering the prototype implementation, while Section 5 presents the evaluation results. After that, Section 6 compares our approach with related work, before the paper closes with a short conclusion.

## 2 LIMITATIONS OF DATA STREAMING INSIDE STATE MACHINE REPLICATION

SMR is a way to actively replicate a service over multiple servers [16, 25]. Each replica runs a service instance that is implemented as a finite state machine, so that all replicas starting in the same state and receiving the same inputs in the same order will eventually reach the same state, having produced the same outputs. A total-order broadcast ensures that client requests are received in the same order at all replicas. In this paper, we focus on quorum-based consensus algorithms like [3, 6, 17], which use a view abstraction for group membership management and a sequencer for request ordering, even if other approaches exist [10]. All participating replicas advance through a sequence of views, each controlled by a single leader replica, which is determined by a leader election protocol as part of the consensus algorithm. The leader replica defines the actual order of the requests. The failures of normal replicas are masked by the replica redundancy. A leader failure requires a view-change, in which another replica is elected as the new leader to continue execution. Redundant request execution enables the comparison of results, to track down Byzantine faults. The state of the state machine is abstracted in a state management component that synchronizes the state between replicas, if new replicas join the system or if a replica has fallen behind and needs to catch up.

Data Streaming platforms maintain a continuous stream of data in order to create real-time data pipelines and high-level stream processing applications [21, 27]. For this purpose, broker nodes store data records in a distributed data log. In Apache Kafka data records are key-value pairs identified by a numerical incrementing offset index. A log is divided into topics (see Figure 1), which in turn are subdivided into one or several ordered sequences of immutable records – so called partitions. Amazon Kinesis on the other hand stores data blobs in streams, which are subdivided in shards, but for this paper we will adhere to the Apache Kafka terminology. Producers publish or produce records by sending them as messages to the responsible brokers, where they will be appended to the partitions. The offset of a partition designates the last committed record. Consumers consume records by constantly polling the brokers with their last received offset of those partitions they are subscribed to. The partitions are stored on the local disk. A retention policy defines for how long records are stored before the partition is truncated from the head.

In order to provide reliability, data streaming platforms use passive replication to replicate partitions over multiple broker nodes. Each partition is assigned to one designated leader and can have multiple followers. The leader commits the appended records to its followers, so that they are able to replace the leader node if it crashes. For scalability purposes, the partitions of one topic can be scattered around multiple brokers and consumed by different consumers in parallel. The broker-partition distributions as well as the leader-follower assignments are stored in a central configuration. Apache Kafka maintains this information in a separate Apache ZooKeeper cluster, which is a distributed hierarchical key-value store.

Protecting services from arbitrary faults and Byzantine attackers by wrapping those services into SMR middleware proved to be a feasible approach with reasonable results for many kinds of services [4, 6, 12, 26]. However, a naive first integration of data streaming with an SMR middleware revealed very poor performance. This is because several SMR concepts limit the abilities of data streaming. These conflicts concern mainly the state management and scaling capabilities.

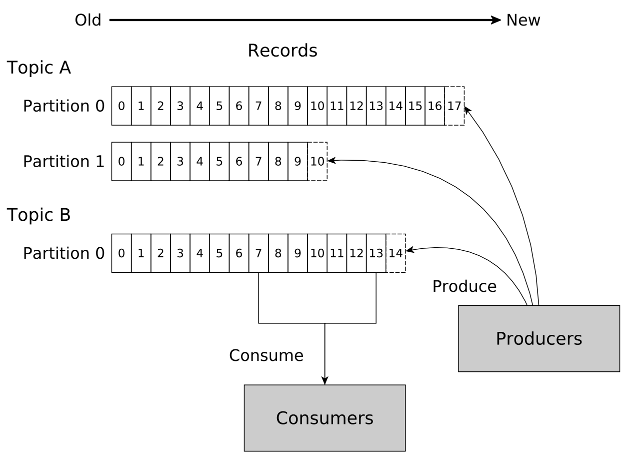

**Figure 1: Data Streaming Log Anatomy: New records are appended by producers at the tail; existing records are retrieved by consumers from their last consumed offset**

## 2.1 State Management

In theory, SMR systems can operate with their protocol and service state purely in volatile memory, as long as the number of faulty replicas is bounded by some $f$ out of a total of $n$ replicas, with typical values of $f \leq (n-1)/2$ in crash fault tolerant and $f \leq (n-1)/3$ in Byzantine fault tolerant systems. Practical SMR middleware, however, should also enable recovery from a larger number of faults, including the temporary crash or shutdown of all replicas. Recovery requires the integration of mechanisms for protocol state logging, checkpointing, and state transfer [2].

In a crash-recovery fault model, algorithms such as Paxos provide strong consistency across any number of crashes by maintaining a locally persistent log of all relevant protocol actions at each replica [8]. To prevent the log from growing infinitely and avoid long time delays when processing log entries, the log is truncated during periodic checkpoints, in which the current service state is stored in a snapshot. These snapshots are exchanged in addition to the information from the logs during state synchronization.

In the Byzantine fault model, Bessani et al. [2] describe a similar approach for recovering from more than $f$ crashes, also based on persistent protocol state logging and checkpointing. Their approach offers *durability*, defined as "the capability of a SMR system to survive the crash or shutdown of all its replicas, without losing any operation acknowledged to the clients". In addition to that, the BFT model makes it necessary that the integrity of the exchanged log entries and the snapshot is verified during recovery. For this purpose, the receiving replica re-evaluates the consensus decisions and only accepts snapshots when it receives $f+1$ matching snapshots from different authenticated replicas [6].

The textbook state handling approach is contrary to the characteristics of data streaming, in which the service state consists of a large, ever-growing data log. This data log is persisted on stable storage, as it usually does not fit into main memory, and due to reliability reasons. It is only modified by produce requests, which consist of log record batches that are appended to the data log during the request execution. So, keeping a history of those requests in the protocol log means that the data written to disk by the streaming service is written to disk a second time for the log. Thus, applying the default SMR state handling to the service state of data streaming results in a massive waste of resources and the management of those redundant state objects affects the overall performance. There is also the risk of memory overflows, if the state management does not calculate the exact resource requirements and adapts the data structures maintaining those state objects accordingly. Traditional checkpointing makes this situation even worse. At recurring times, the entire data log must be read from the application once again to be stored as a checkpoint. It is therefore obvious that an efficient state handling must reflect the request and state characteristics of data streaming as well as its persistent storage.

## 2.2 Scalability

Quorum-based SMR suffers from limited scalability. Horizontal scaling (which usually means adding new machines to the system) is not an option, as the strong consistency property requires all correct replicas to agree on a global request order and to execute all requests. During the agreement phase, the voting steps of the protocol often require a quadratic number of message exchanges, so that the communication overhead grows rapidly with each additional replica. The execution phase does also not benefit from additional replicas, since all replicas have to execute every request. Vertical scaling can provide some performance improvements [13], but is also limited by the maximum resources per replica.

Data streaming platforms, however, utilize their highly partitioned service state to implement efficient horizontal scaling. Therefore, the partitions of large topics are distributed over different broker nodes and processed in parallel. When data streaming is combined with SMR this benefit can be maintained and the scaling limitations of SMR can be circumvented as long as the scaling is applied on a cluster level. Therefore, the disjoint partitions of large topics are distributed over multiple replica groups, which enables parallel processing of those topics. In this case, however, the distribution and synchronization of the assignments between partitions and replica groups must be ensured by the system through additional means.

## 3 SMARTSTREAM

In this section we propose our solution for a resilient and scalable data streaming platform – SmartStream. We first state our assumptions on the system model and define the provided service guarantees, before we give an architectural overview of the components and explain their interaction. After that, we discuss the core concepts in more detail.

## 3.1 System Model and Definitions

SmartStream is a distributed system with an unbounded number of client nodes $C = \{c_1, c_2, ...\}$ and a fixed number $n$ of server nodes $S = \{s_1, ..., s_n\}$. Clients and servers are connected by an unreliable network and communicate with each other by message passing. The network may drop, duplicate or deliver messages out of order, but common techniques like message authentication and packet retransmission may mask this unreliability. The system is partially synchronous, so there are bounds on message delays or relative processing speed, but they are not known and hold only after some time.

Nodes can be either correct or faulty. Correct nodes deterministically follow the algorithm specification. Faulty nodes can either crash, or act in an arbitrary or maliciously Byzantine way. The system maintains safety and liveness as long as only $f = \lfloor \frac{n-1}{3} \rfloor$ nodes are faulty. The system can maintain state consistency even if more than $f$ nodes crash, but liveness can only be maintained as long as the number of faulty (including crashed) nodes is not greater than $f$ or as soon as it not more than $f$ again. In the presence of malicious Byzantine faults, we assume a strong attacker that has full control over the faulty nodes and is able to impersonate them. The attacker is able to access the key material of controlled nodes but is not able to break the cryptography primitives of correct nodes.

## 3.2 System Architecture

SmartStream follows the modular approach illustrated in Figure 2. Dashed arrows show message-based communication flows between components, whereas solid arrows depict direct component interaction. The foundation is a communication module that provides reliable and authenticated communication channels between nodes using a reliable transport protocol and message authentication based on message authentication codes (MACs) or signatures. A modular SMR protocol is built on top of the communication module. The SMR protocol uses a consensus algorithm to implement the atomic broadcast. We assume a quorum-based consensus algorithm with a view-reigning sequencer like [6]. This algorithm produces verifiable and therefore reproducible decisions. A view manager is managing the replica group membership, and a state manager is maintaining the state.

On top of the SMR protocol operates the data streaming service. The data log manager maintains a stream of data log entries and provides interfaces for producing and consuming these records. The cluster manager maintains the distribution of the data log over the replica groups. In traditional SMR libraries, the service layer is usually conceptually separated from the SMR protocol, but in order to overcome the limitations mentioned in Section 2 we had to introduce several dependencies in the form of component interaction between cluster and view manager as well as data log and state manager.

## 3.3 Data Streaming Service

Figure 3 displays a SmartStream cluster during producing and consuming. The view and cluster managers are omitted for better readability. The service is actively replicated over multiple broker replicas using the state machine approach. Clients are either producers that publish data into topics, or consumers that consume data from subscribed topics. Data is stored as records in the distributed data log, managed by the log manager and structured in topics and partitions. The producers and consumers are not replicated. Therefore their fault tolerance needs to be handled by the application using the data streaming platform.

For producing records, producers send a list of records in a produce request to all broker replicas. Records carry the target topic and partition as well as the actual data as key-value pairs. The leader replica then orders the requests by initiating the consensus protocol. As soon as the consensus protocol decides the proposed order, the log manager can execute the request by appending the records to the respective partitions. The updated offsets of the involved partitions are stored in the offset log of the state manager and also sent as response to the producer. The producer will consider the records committed as soon as it receives $f + 1$ matching responses.

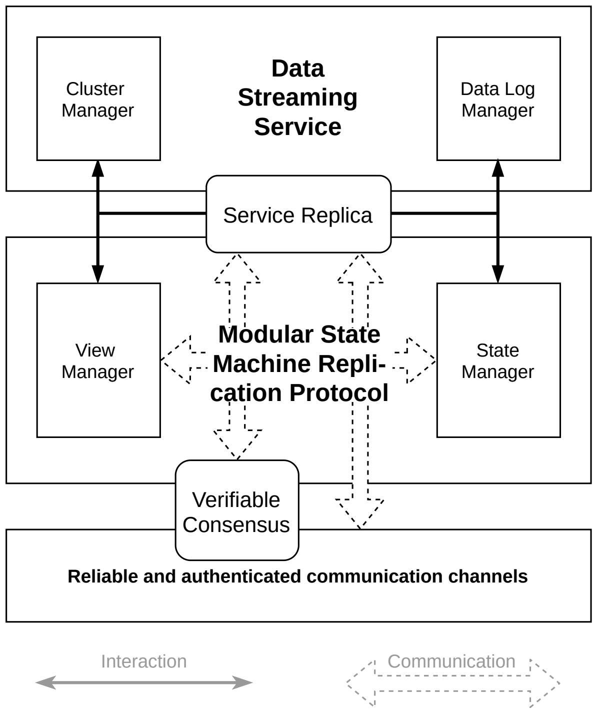

**Figure 2: SmartStream Modules: Introduced adaptions require interaction between the service and middleware layer**

For consuming, consumers can subscribe themselves to one or multiple partitions. The subscription process starts a consume session. With the poll operation the consumers query the brokers for unread records. Therefore, the consumers send along their last consumed offset for every subscribed partition. As those fetch requests do not alter the service state, those requests do not need to be ordered at the brokers, but can be answered directly with the matching records. Consumers accept these records as soon as they receive $f + 1$ matching responses. During a consume session, SmartStream ensures that consumers consume each log record only once.

## 3.4 State Management

To avoid the disadvantages stated in Section 2 and to exploit the append-only semantics of data streaming, SmartStream uses a lightweight but sophisticated state management. The general idea is to get rid of the protocol log for executed requests in order to prevent redundant storage of data log records, and to support incremental state updates for efficient state handling.

To avoid having to maintain the protocol log, we simply generate a new checkpoint after the execution of the requests decided in the

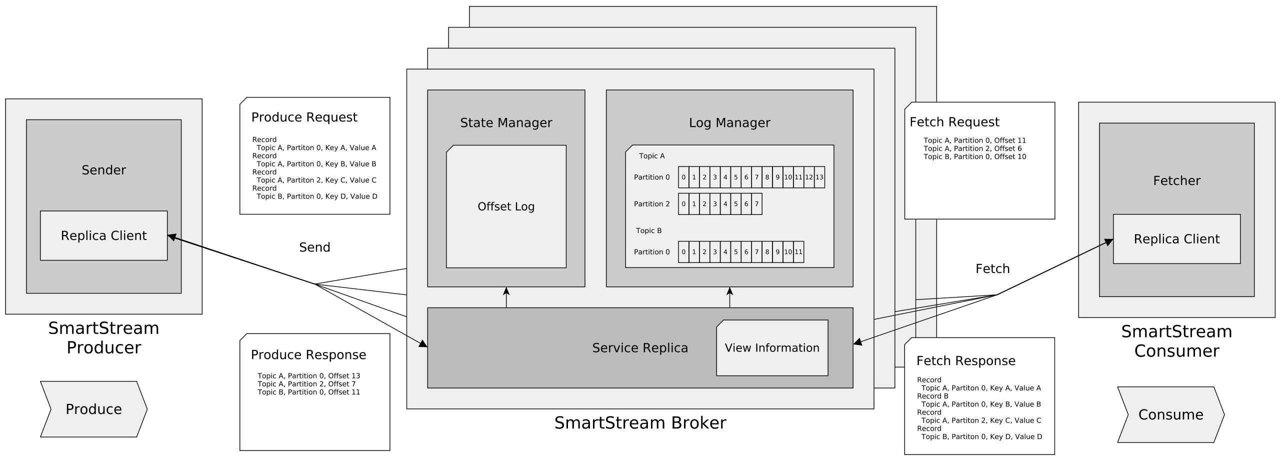
Sender
Replica Client
SmartStream Producer
Produce
Produce Request
Record
Topic A, Partiton 0, Key A, Value A
Record
Topic A, Partiton 0, Key B, Value B
Record
Topic A, Partiton 2, Key C, Value C
Record
Topic B, Partiton 0, Key D, Value D
Send
Produce Response
Topic A, Partition 0, Offset 13
Topic A, Partition 2, Offset 7
Topic B, Partition 0, Offset 11
State Manager
Offset Log
Log Manager
Topic A
Partition 0
Partition 2
Topic B
Partition 0
Service Replica
View Information
SmartStream Broker
Fetch Request
Topic A, Partition 0, Offset 11
Topic A, Partition 2, Offset 6
Topic B, Partition 0, Offset 10
Fetch
Fetch Response
Record
Topic A, Partiton 0, Key A, Value A
Record B
Topic A, Partiton 0, Key B, Value B
Record
Topic A, Partiton 2, Key C, Value C
Record
Topic B, Partiton 0, Key D, Value D
Fetcher
Replica Client
SmartStream Consumer
Consume

**Figure 3: SmartStream Message Flow: Producers send records to the replicated brokers; consumers fetch records from them**

particular consensus instance. Usually the creation and persistence of a state snapshot is a costly operation, but the persisted data log already carries the snapshot contents and there is no need for further persistence. From that follows that the protocol messages are only needed within the currently running consensus instances and must only be kept until the respective requests have been executed. The ability of view changes is not influenced by immediate checkpoints. During a view change, the responsibility of a new leader is to synchronize the cluster to the most recently executed consensus, finish already started consensus instances and to propose any unanswered client request. The latter two tasks concern the protocol before request execution and are therefore not affected by our optimization. The synchronization to the latest executed consensus can also be performed by exchanging the snapshot of the respective consensus instance. At least $f + 1$ correct replicas must be able to provide this snapshot, as a checkpoint is only created if a quorum of $2f + 1$ replicas have committed themselves to execute the corresponding requests. With that simplification we fulfill our first goal.

Relying on a full state exchange during each state transfer is not a practical assumption. Therefore, we further need to optimize our state transfer in a way to support incremental state updates. This service state partitioning is inspired by prior work [6, 20], but also leverages the append-only semantics and data log storage characteristics to simplify the process. For this purpose the state manager keeps a history of the checkpoints and stores for every consensus instance $i$ the offset list of its $n$ partitions $o_i := (o_{p1}, o_{p2}, ..., o_{pn})$ (see Figure 4). If a replica queries the state starting from its last known consensus instance up to the current consensus instance, the other replicas can fetch exactly those records from the data log. The state manager of SmartStream is hereby aware of the data log structure and is able to interact with the data log through its application interface. With this functionality we satisfy our second objective.

The checkpoint log is implemented as a ring buffer, to prevent it from growing infinitely. If a replica requests a state transfer and submits a last known checkpoint that is no longer in the buffer

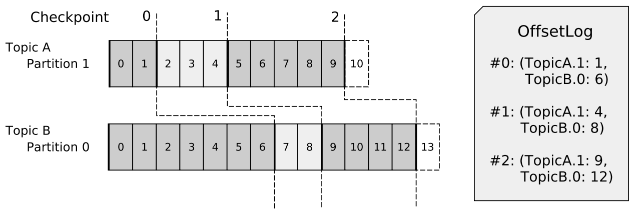
Checkpoint
0
1
2
Topic A Partition 1
Topic B Partition 0
OffsetLog
#0: (TopicA.1: 1, TopicB.0: 6)
#1: (TopicA.1: 4, TopicB.0: 8)
#2: (TopicA.1: 9, TopicB.0: 12)

**Figure 4: SmartStream Offset Log: Each checkpoint stores the current data log offsets**

or there is no last known checkpoint at all, a full state transfer is needed. In this case the complete log is transferred using the segment files stored on disk. To satisfy the durability property, the log manager keeps books about the responses sent to the client. In case of a disaster recovery it has to synchronize its persistent state with the other replicas to make sure that all replies that have been received by clients are reflected in the replica state.

## 3.5 Cluster Configuration

In order to utilize the partitionable state structure of data streaming and to provide scalability as described in Section 2, SmartStream has to solve three challenges: (i) managing multiple replica groups in one cluster, (ii) the dynamic assignment of partitions to replica groups, and (iii) the transfer of partitions during reassignments. Therefore, SmartStream makes use of a special view management client that is trusted by all replicas and can be used to dynamically reconfigure the cluster at runtime, as described in [3].

In general, the extended SmartStream cluster consists of one or more replica groups $G = \{g_1, ..., g_m\}$, each forming its own and independent SMR group. The service state is divided into a set of partitions $P = \{p_1, ..., p_p\}$. Each partition is always assigned to exactly one replica group using a list of assignments $A = \{a_1, ..., a_a\}$. To solve the first two problems, an extension of the view component is necessary. The view usually holds information about the current regency $r$ as well as its current members $g$ and is numbered consecutively $v_i = (r_j, g_k)$. In order to reflect the other replica groups and the global assignments, the view component is transformed into a tuple, which still holds the local regency information $r$ but also the current set of replica groups as well as the currently valid set of assignments $v_i = (r_j, G_k, A_l)$, and is identified by the vector $(j, k, l)$. An example of the extended view component for a cluster with multiple replica groups is illustrated in Figure 5. Changes in the regency are triggered by the view-change protocol of the consensus algorithm and affect only the local replica group, whereas changes in the cluster or assignment configuration are carried out by the trusted client and affect all replica groups.

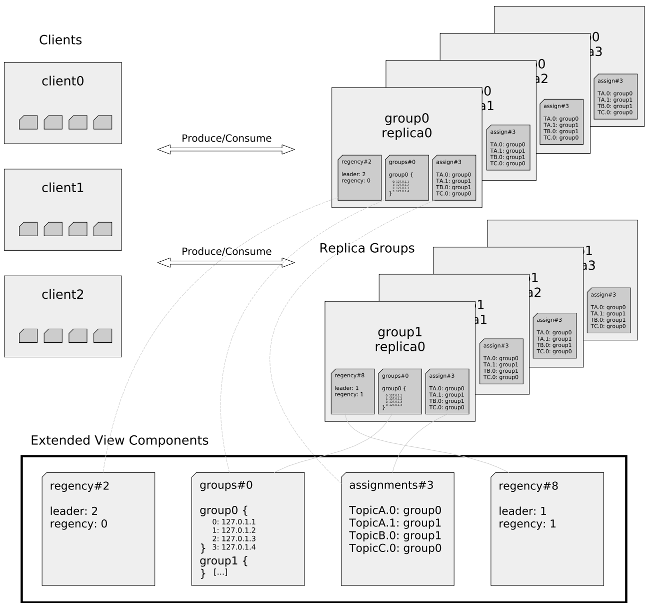

**Figure 5: SmartStream Replica Groups: Clients are directed by means of the extended view components**

The client can carry out operations to add or remove replicas and replica groups as well as to add, delete or transfer partitions. Those operations are sent to each replica group and are internally ordered. Changes to the cluster or assignment configurations increment their respective identifiers, so that the current view identifier can ensure view freshness at the clients. Clients store local copies of the views and send their last known identifiers with every request, so that all replicas can signal outdated view information to them. The local view copies can be used to lookup the replicas responsible for a partition whenever they initiate a request ($lookup : P \rightarrow G$).

For the last issue, we make use of a partition transfer protocol for replicated state machines that also uses a trusted view management client, as described in [22]. The slightly modified protocol is executed as follows: (i) the trusted client $c$ contacts the replica group $g_o$ responsible for the partition $p$ and indicates the new replica group $g_n$. (ii) group $g_o$ stores the current offset $i$ of the partition $p$ and invokes an inter-group partition transfer that is a variation of the state transfer and therefore resistant against malicious replicas. (iii) as soon as the transfer is acknowledge by group $g_n$, group $g_o$ starts another incremental partition transfer with the records starting from offset $i$ and starts redirecting all requests for partition $p$ to group $g_n$. (iv) as soon as group $g_n$ informs the client $c$ about the finished transfer, client $c$ commits the new assignment $a$ at all replica groups of the cluster $G$. $g_o$ then stops answering the requests for $p$.

## 3.6 Optimizations

The following optimizations are used to further improve the performance of SmartStream.

*3.6.1 Batching.* Batching multiple client requests together helps to reduce the overhead of the latency-intensive consensus protocol. This is especially important in the case of data streaming, as every consensus decision involves disk access, and sequential writes usually outperform random writes. Therefore SmartStream batches the requests already in the producers before aggregating those requests again within the SMR protocol. The producers sort the requests by responsible replica groups. When request forwarding between replica groups is active in case of a partition transfer, there is a risk that the messages are delayed and time out. The impact is dependent on the exact timeout intervals and the frequency of such transfer operations. However, we consider the expected disturbances to be tolerable, since partition transfers should be a rather rare operation.

*3.6.2 Unsynchronized reads.* In SMR systems, read requests, which do not alter the service state, do not need to be ordered but can be answered directly. A client accepts a read response as soon as it received the same response $f + 1$ times. Thus it can rule out the presence of $f$ wrong answers. However, whenever the request processing of the servers is not exactly synchronous – which can easily happen in wide-area setups – the quorum may not be reached, as the answers may refer to different service states even if they are correct. To resolve the situation, the client retransmits the message and explicitly insists on the request to be ordered to ensure that the response quorum is met.

This latency intense retransmission can be avoided with the append-only data log in the case of data streaming. Hereby, the client is actually able to resolve such conflicts on its own by accepting as valid all successive records that appear $f + 1$ times in the responses. All records that do not satisfy this condition need to be fetched again with the next poll operation. It has to be noted that clients first check the validity of the entire message. If that fails, they need to validate each of the individual records successively. Thus, the additional record verification places an additional burden on the client. However, if the sequential ordering represents the bottleneck of the SMR system, this optimization can improve the overall performance.

## 4 IMPLEMENTATION

This section describes the implementation of our SmartStream prototype. As the implementation from scratch is very time-consuming, we decided to make use of existing solutions. This way we adapted some Apache Kafka components to be able to run on top of the BFT-SMaRt middleware [3]. Our implementation is based on Apache Kafka 2.13.2 and BFT-SMaRt 1.2. We tried to leave the main functionality of Apache Kafka untouched. The log anatomy as well as the message formats are the same as in the original version. Instead of implementing the full Kafka API, we limited ourselves to producing and consuming of records as well as exchanging metadata. Higher-level APIs such as the management of access restrictions or the rebalancing of consumer groups have been omitted. The current prototype does not satisfy the durability property to survive concurrent crashes of more than $f$ replicas.

For the client side we implemented custom SmartStream producers and consumers that extend the given Kafka producers and consumers. The original producers and consumers are equipped with sender and fetcher components, which communicate with the remote brokers via a network client using non-blocking IO. We created our own implementations for the sender and fetcher, which utilize BFT-SMaRt clients to communicate with the BFT-SMaRt servers inside the SmartStream brokers. Those components also use non-blocking IO, carried out through the Netty communication framework.

For the server side we implemented custom SmartStream brokers, which are in fact BFT-SMaRt servers that mimic the Kafka API calls. Each broker controls one original data log manager component that writes and reads records from and to disk during producing and consuming. We further provided a custom state manager that is able to handle the optimized state management. The direct communication between the replicas eliminates the ZooKeeper cluster completely. As the topic metadata is also stored in the ZooKeeper cluster we needed to transfer this information back into the view management. We extended the Kafka node implementation to fit our BFT-SMaRt clusters into this data object and adapted the metadata mechanics, which are used for the Kafka cluster configuration. A BFT-SMaRt cluster is now represented as a Kafka node, which allows the assignment of partitions to replica groups. The discovery of the servers by clients is done with the default BFT-SMaRt configuration files. However, the configuration is now done on a group-level in order to support multiple replica groups inside the cluster.

## 5 EVALUATION

This section presents the evaluation of our prototype implementation. We start with the definition of suitable reference measurements and the description of our test environment, before we illustrate and discuss the results of our performance evaluation. We executed microbenchmarks that mimic the official performance benchmarks shipping with Apache Kafka.

### 5.1 Evaluation Setup

All tests were executed in the Amazon EC2 cloud on M5ad general purpose instances. The broker nodes are placed in xlarge instances, which each have four virtual CPU cores and 16 GB of RAM as well as a 150 GB local NVMe SSD with an average sequential write performance for a 1GB large file around 144 MB/s. The network performance is limited to 10 GBit/s. The write performance is a limitation of the specific instance type. There are other instance types with higher disk throughput available – even the network-attached elastic block store volumes can reach higher throughput rates – however, with the selected instance type the maximum throughput reachable during the evaluation is limited by the local disk and not by network performance. Producer nodes are placed in 4xlarge instances, which have 16 virtual CPU cores and 64 GB of RAM. We choose this instance type as we place multiple producers in one VM. The workload is carried out using a central coordinator component to activate and deactivate the producers.

| Name | Fault Mode | Nodes | Faults |
|---|---|---|---|
| KafkaCft0 | CFT | $n = 1$ (+1) | $f = 0$ |
| KafkaCft1 | CFT | $n = 3$ (+3) | $f = 1$ |
| SmartStreamCft1 | CFT | $n = 3$ | $f = 1$ |
| SmartStreamBft1 | BFT | $n = 4$ | $f = 1$ |
| KafkaCft2 | CFT | $n = 5$ (+5) | $f = 2$ |
| SmartStreamCft2 | CFT | $n = 5$ | $f = 2$ |
| SmartStreamBft2 | BFT | $n = 7$ | $f = 2$ |

**Table 1: System Variants: Cluster configurations used during performance evaluation**

Table 1 shows the different clusters' configurations, which differ in the configured fault mode, the number of broker nodes and the resulting number of tolerated faults. In order to asses the effectiveness of our approach we included reference measurements with the original Apache Kafka implementation — one base measurement with an unreplicated broker and two measurements for each number of tolerated faults respectively. We configured Apache Kafka to provide crash fault protection similar to that provided by SmartStream. This means the cluster only replies if records have been fully committed at the backup nodes and that there are always enough synchronized backup nodes to take over in the presence of faults. This results in a cluster size of $2f + 1$ nodes. The original Kafka needs an additional ZooKeeper Cluster of $2f + 1$ nodes. We placed those nodes in dedicated servers, even if they might be co-located with the broker nodes in practice. Since SmartStream can be used in both CFT and BFT mode, we performed the measurements in both fault modes to see what effects cluster and quorum sizes have on performance.

It has to be noted that apart from the additional hardware costs introduced by the additional $f$ replicas, the BFT variant of SmartStream also has a significantly higher bandwidth usage, as during producing the records are sent to all replicas and during consuming all records are received from all replicas. This means that the bandwidth costs rise from $n$ for producing and 1 for consuming in the original Kafka to $2n$ for producing and $n$ for consuming in SmartStream.

### 5.2 Performance Evaluation

The performance evaluation measures the performance of the different system variants in the form of throughput and latency. From the average disk throughput of the producer instances we derived the following producer workload. We place 128 producers in one instance. Each of those 128 producers produces up to 1000 records per second, each consisting of an empty key and a 1024 Byte long random value as well as an 8 byte long timestamp. The records are sent with an intermediate delay of 1 ms, without blocking while waiting for the response. Not considering additional storage overhead

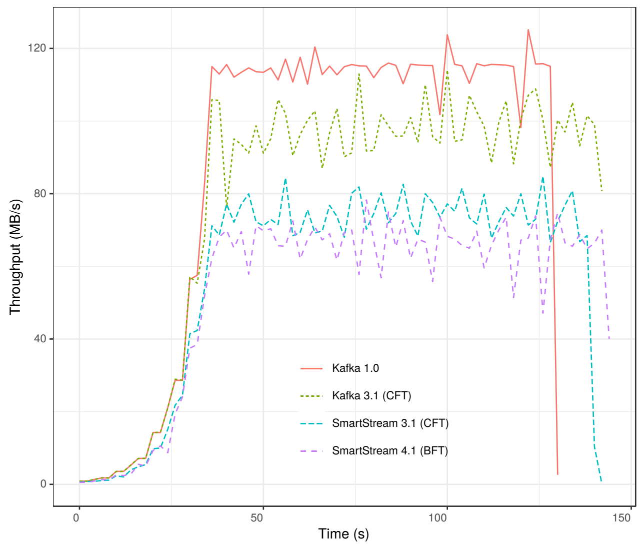

**Figure 6: Throughput Performance: Peak throughput performance for system variants tolerating one fault**

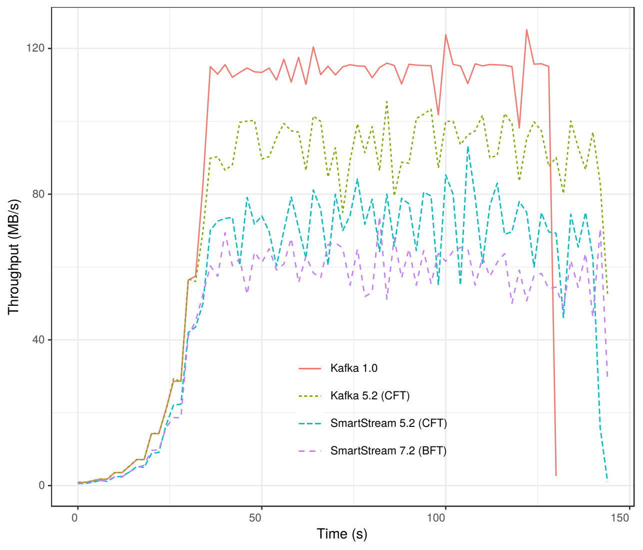

**Figure 7: Throughput Performance: Peak throughput performance for system variants tolerating two faults**

and processing times, this results in a theoretical peak producer throughput of 126 MB/s, which is enough to keep the brokers busy.

The actual workload starts with one producer and doubles the number of active producers every five seconds, resulting in 128 active producers per VM after 40 seconds. We then keep these producers active for 90 seconds, before we stop all producers from sending for further ten seconds to finish queued up requests. Figure 6 shows the actual throughput results over time. As expected, the unreplicated variant performs best with a maximum throughput of nearly 120 MB/s. This ideal rate drops for the replicated Kafka variant by 20 MB/s and fluctuates around 100 MB/s. The SmartStream variants then lose another 20 (CFT) or 30 (BFT) percent in performance. Figure 7 draws a similar picture for the variants with two tolerated faults. All variants (except for the reference measurement) experience a throughput drop by roughly 10 MB/s.

Those measurements show the maximum throughput rates, possible during request peaks. However, if the loads would be kept at those rates for long times, produce requests would pile up at the producers as well as inside the brokers and a noticeable amount of requests would eventually time out. Handling those requests would harm the overall system performance further. Therefore, we also measured the request latency at the producers for the workloads from above. The results are shown in Figure 8. It can be seen that latency rises at certain throughput rates. The Byzantine fault tolerant SmartStream variant experiences rising latency values at around 47 MB/s, the crash fault variant at around 50 MB/s. For the replicated Kafka variant we see increased latencies at around 70 MB/s. Only the unreplicated Kafka stays responsible during the whole experiment. The latency jumps are caused by the fact that the messages are not processed fast enough and therefore remain in queues for longer.

We also carried out one experiment with synchronous producers. These producers only send requests if they received the response

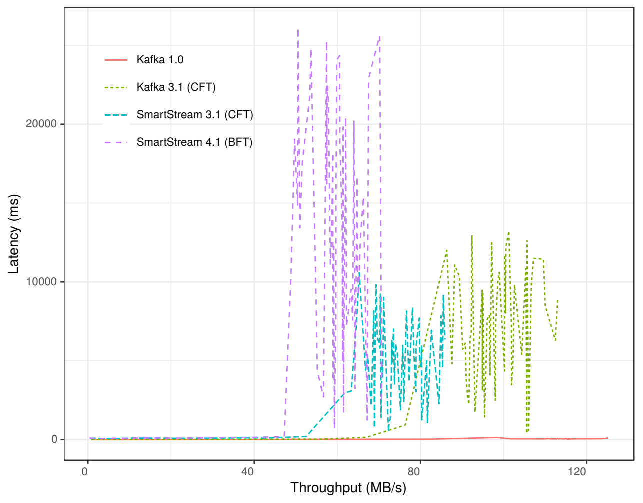

**Figure 8: Latency Performance: Rising graphs show the times at which the system performance saturates**

for the previous request. We placed 90 producers in four producer VMs, resulting in a total of 360 producers. The workload starts with 20 producers and adds an additional 20 producers every five seconds, running for a total time of 90 seconds. The latency to throughput comparison is shown in Figure 9. With this harmonized sending behavior, all variants can maintain a low latency at least until throughput reaches 60 MB/s. This underlines the influence of proper workload shaping on the overall system performance.

In general it can be concluded that SmartStream suffers a moderate performance drop. While there is no significant difference in delay during periods of moderate workloads, the drop in throughput can be up to 30 percent at peak loads. This overhead can be

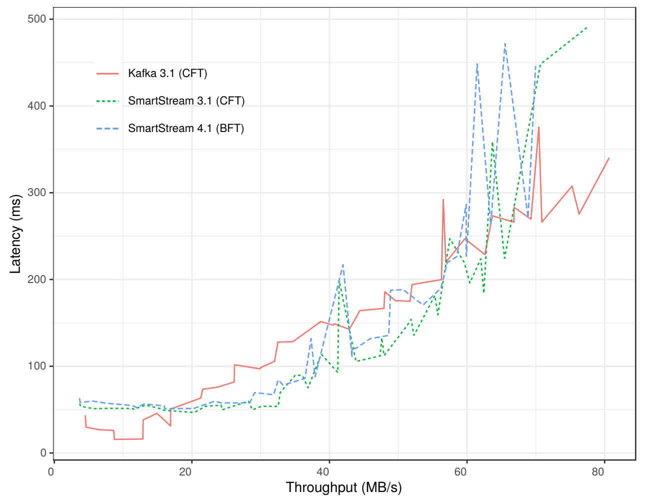

**Figure 9: Latency Performance: Synchronously sending producers utilize system capacities more efficiently**

explained by the increased complexity of the algorithm and the extended communication of the additional nodes. The scalability properties of SmartStream ensure that the proven performance can be maintained even with increasing demands.

## 6 RELATED WORK

This section presents related work in the context of Byzantine fault-tolerant publish-subscribe concepts and scalable SMR approaches.

### 6.1 Byzantine Fault-Tolerant Publish-Subscribe

The preliminary work in the area of Byzantine fault-tolerant publish-subscribe appears to be limited. The authors of [9] identify the tradeoff between reliability and scalability as the hardest challenge and sketch two possible approaches to provide a BFT broker system. The first is a single centralized broker, which is replicated using SMR, and the second is a network of brokers, which meets the reliability requirements by replicating nodes and connections in an application-specific overlay network. In general, the authors attribute to the former a simpler structure and to the latter better scalability. However, they do not provide a system description or protocol specification for either of their approaches.

A first real solution using an overlay of replicated brokers is PubliyPrime, which exploits the overlay network to identify and exclude Byzantine brokers [15]. The tree-based overlay network stores multiple intermediate brokers as a path from the publisher to the subscriber. Messages are sent redundantly over multiple of these paths. In case one of those broker nodes is suspected to act Byzantine, its successor nodes are directly consulted for message dissemination. By verifying the message digests of received messages and by constantly checking whether all obligated brokers forwarded the required messages, suspicious nodes are identified. A similar approach sketches the organization of brokers in chains on a tree-based overlay network [14]. Publications are forwarded following the chains until a quorum of brokers accepts the publication and sends it to the respective subscriber. In order to minimize the number of forwarded messages the protocol runs a weak consensus in fault-free execution, whose history is verified with a reliable consensus in periodic verification rounds.

Other approaches implement a centralized replicated broker using the SMR paradigm. Trinity is a more recent approach with a centralized broker [24]. It combines multiple pluggable blockchain platforms (Tendermint, HyperLedgerFabric, IOTA, Ethereum) for message ordering and storage with an MQTT broker for publish-subscribe message dissemination. The use of public blockchains results in high message delays of several seconds, which are arguably only usable in select use cases. The permissioned blockchain variants also leave a lot of room for improvement, as the message delay varies between 30 and 140 ms at 30 transactions per second – which we consider very high for a local area network. Real-world challenges like log retention or state management are not covered at all.

### 6.2 State Management in State Machine Replication

For services with large states, state management very quickly becomes a limiting factor. More efficiency can be achieved by either optimizing the state transfer or by partitioning and distributing the state over multiple groups. In order to support incremental state transfers, various approaches organize the state in Merkle trees or custom tree-based data structures that are inspired by them [6, 20]. Other approaches leverage their application-specific characteristics. The authors of [28] (re-)integrate new or outdated replicas into the view without performing a complete state update. Replicas know which state objects are needed for executing certain requests and fetch only exactly those state objects on-demand during request execution. The remaining state is fetched by those replicas in the background eventually. [11] exploits the capabilities of hypervisors in order to rapidly get the current application state as a snapshot volume. One volume copy of the snapshot can be used for the state transfer, whereas a second copy-on-write volume can be directly used to continue request processing.

Other approaches utilize state partitioning to improve scalability. Augustus is a scalable BFT key-value store that distributes the application state over multiple replica groups [23]. The keys are distributed over the partitions using a deterministic range partitioning scheme or deterministic hashing, which allows clients to address the responsible replica groups. Single-partition commands can be executed directly, whereas multi-partition commands are executed using an atomic commit protocol with locking. Main drawbacks are the transaction overhead and the inflexibility of the static partitioning scheme. A similar approach for CFT replicated state machines is proposed in S-SMR [5]. In this approach the whole state is dissected in state objects that are assigned to various partitions. The assignment is administrated by a user-defined oracle, which can be consulted to learn the responsible replica groups of the state objects affected by client commands. Single-partition commands are again trivial. During multi-partition commands, the state objects are forwarded from the responsible replica groups to the involved replica groups, so that all replica groups are provided with local copies of state objects that are not maintained by their partitions. The approach requires a static assignment between state objects and partitions as well as additional code for exchanging or incrementally updating the local state object copies. The static assignment drawback is addressed in DS-SMR [18] with a dynamically adapting partitioning scheme that takes the individual partition workload into account. During multi-partition commands, the refined scheme always migrates all state objects of one command to a new partition in order to leverage data locality and improve the overall performance. A partition transfer protocol that allows the transfer of state partitions between replica groups is proposed in [22]. It evaluates the transfer of large state partitions and provides a modular protocol that can be applied to BFT SMR protocols. In fact we integrated a variant of this protocol into our approach.

## 7 CONCLUSION

This paper presented a Byzantine fault tolerant data streaming platform based on SMR. It can be concluded that data streaming can be ideally combined with SMR due to the partitionable state structure and the strictly monotonously growing system state, if certain adjustments are applied. With the optimized state management and scalable cluster scheme we proposed simple but effective solutions that exploit these optimization potentials and can be adapted to other systems with similar characteristics. The provided implementation reaches good performance in the Byzantine fault model, even though throughput is reduced by up to 30 percent at peak loads. However, the also introduced horizontal scaling capabilities can mitigate this weakness through the use of additional computing resources. In addition, performance losses as a consequence of increased resilience can arguably be tolerated. In fact, to the best of our knowledge, SmartStream represents the first practically usable Byzantine fault tolerant data streaming platform that goes beyond early research prototypes, which are not considering important functionalities like state management and log maintenance. We are certain that there is still further potential for performance improvements in the implementation, even if we showed that the platform can be of practical relevance and does not need to hide behind its crash fault tolerant counterparts. Also, the detailed partition distribution and batch tuning, which are important factors for reaching good performance, would benefit from an automated and self-adjusting approach, as the request characteristics and environmental conditions may change over time. We leave this open as future work.

## ACKNOWLEDGMENTS

This work has been funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 268730775 (OptSCORE).

## REFERENCES

[1] Amazon Web Services LLC. 2008 (accessed Sep 28, 2020). *Amazon S3 Availability Event: July 20, 2008.* https://status.aws.amazon.com/s3-20080720.html

[2] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. 2013. On the Efficiency of Durable State Machine Replication. In *2013 USENIX Annual Technical Conference (USENIX ATC 13).* 169–180.

[3] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. 2014. State machine replication for the masses with BFT-SMaRt. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on.* IEEE, 355–362.

[4] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. 2008. DepSpace: a Byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008.* 163–176.

[5] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. 2014. Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks.* IEEE, 331–342.

[6] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)* 20, 4 (2002), 398–461.

[7] Frederico Cerveira, Raul Barbosa, Henrique Madeira, and Filipe Araújo. 2020. The Effects of Soft Errors and Mitigation Strategies for Virtualization Servers. *IEEE Transactions on Cloud Computing* (2020).

[8] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC '07).* ACM, 398–407. https://doi.org/10.1145/1281100.1281103

[9] Tiancheng Chang and Hein Meling. 2012. Byzantine fault-tolerant publish/subscribe: A cloud computing infrastructure. In *2012 IEEE 31st Symposium on Reliable Distributed Systems.* IEEE, 454–456.

[10] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)* 36, 4 (2004), 372–421.

[11] Tobias Distler, Rüdiger Kapitza, and Hans P. Reiser. 2008. Efficient state transfer for hypervisor-based proactive recovery. In *Proceedings of the 2nd workshop on Recent advances on intrusiton-tolerant systems.* 1–6.

[12] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguiça. 2011. Efficient middleware for Byzantine fault tolerant database replication. In *Proceedings of the sixth Conference on Computer systems (EuroSys'2011).* 107–122.

[13] Gerhard Habiger, Franz J. Hauck, Johannes Köstler, and Hans P. Reiser. 2018. Resource-Efficient State-Machine Replication with Multithreading and Vertical Scaling. In *2018 14th European Dependable Computing Conference (EDCC).* IEEE, 87–94.

[14] Leander Jehl and Hein Meling. 2013. Towards Byzantine fault tolerant publish/-subscribe: A state machine approach. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems.* ACM, 5.

[15] Reza Sherafat Kazemzadeh and Hans-Arno Jacobsen. 2013. *PubliyPrime: Exploiting Overlay Neighborhoods to Defeat Byzantine Publish/Subscribe Brokers.* Technical Report. TR University of Toronto.

[16] Leslie Lamport. 1978. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)* 2, 2 (1978), 95–114.

[17] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169.

[18] Long Hoang Le, Carlos Eduardo Bezerra, and Fernando Pedone. 2016. Dynamic scalable state machine replication. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN).* IEEE, 13–24.

[19] Adrien Ledeul, Alexandru Savulescu, Gustavo Segura Millan, and Bartlomiej Styczen. 2019. Data Streaming With Apache Kafka for CERN Supervision, Control and Data Acquisition System for Radiation and Environmental Protection. In *17th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'19).*

[20] Barbara Liskov and James Cowling. 2012. *Viewstamped replication revisited.* Technical Report MIT-CSAIL-TR-2012-021. MIT.

[21] Dung Nguyen, Andre Luckow, Edward Duffy, Ken Kennedy, and Amy Apon. 2018. Evaluation of highly available cloud streaming systems for performance and price. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).* IEEE, 360–363.

[22] Andre Nogueira, Antonio Casimiro, and Alysson Bessani. 2017. Elastic state machine replication. *IEEE Transactions on Parallel and Distributed Systems* 28, 9 (2017), 2486–2499.

[23] Ricardo Padilha and Fernando Pedone. 2013. Augustus: Scalable and robust storage for cloud applications. In *Proceedings of the 8th ACM European Conference on Computer Systems.* 99–112.

[24] Gowri Sankar Ramachandran, Kwame-Lante Wright, Licheng Zheng, Pavas Navaney, Muhammad Naveed, Bhaskar Krishnamachari, and Jagjit Dhaliwal. 2019. Trinity: A Byzantine Fault-Tolerant Distributed Publish-Subscribe System with Immutable Blockchain-based Persistence. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC).* IEEE, 227–235.

[25] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)* 22, 4 (1990), 299–319.

[26] Ali Shoker and Jean-Paul Bahsoun. 2012. Towards Byzantine resilient directories. In *2012 IEEE 11th International Symposium on Network Computing and Applications.* IEEE, 52–60.

[27] Guozhang Wang, Joel Koshy, Sriram Subramanian, Kartik Paramasivam, Mammad Zadeh, Neha Narkhede, Jun Rao, Jay Kreps, and Joe Stein. 2015. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1654–1655.

[28] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. 2011. ZZ and the art of practical BFT execution. In *Proceedings of the sixth conference on Computer systems.* ACM, 123–138.