HAL
open science

Virtual Log-Structured Storage for High-Performance Streaming

Ovidiu-Cristian Marcu, Alexandru Costan, Bogdan Nicolae, Gabriel Antoniu

To cite this version:

Ovidiu-Cristian Marcu, Alexandru Costan, Bogdan Nicolae, Gabriel Antoniu. Virtual Log-Structured Storage for High-Performance Streaming. IEEE Cluster 2021, Sep 2021, Portland / Virtual, United States. hal-03300796v1

HAL Id: hal-03300796

https://inria.hal.science/hal-03300796v1

Submitted on 27 Jul 2021 (v1), last revised 30 Jul 2021 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

Virtual Log-Structured Storage for High-Performance Streaming

Ovidiu-Cristian Marcu
University of Luxembourg
ovidiu-cristian.marcu@uni.lu

Alexandru Costan
University of Rennes, Inria
alexandru.costan@irisa.fr

Bogdan Nicolae
Argonne National Laboratory
bogdan.nicolae@acm.org

Gabriel Antoniu
University of Rennes, Inria
gabriel.antoniu@inria.fr

***Abstract*—Over the past decade, given the higher number of data sources (e.g., Cloud applications, Internet of things) and critical business demands, Big Data transitioned from batch-oriented to real-time analytics. Stream storage systems, such as Apache Kafka, are well known for their increasing role in real-time Big Data analytics. For scalable stream data ingestion and processing, they logically split a data stream topic into multiple partitions. Stream storage systems keep multiple data stream copies to protect against data loss while implementing a stream partition as a replicated log. This architectural choice enables simplified development while trading cluster size with performance and the number of streams optimally managed. This paper introduces a shared virtual log-structured storage approach for improving the cluster throughput when multiple producers and consumers write and consume in parallel data streams. Stream partitions are associated with shared replicated virtual logs transparently to the user, effectively separating the implementation of stream partitioning (and data ordering) from data replication (and durability). We implement the virtual log technique in the KerA stream storage system. When comparing with Apache Kafka, KerA improves the cluster ingestion throughput (for replication factor three) by up to 4x when multiple producers write over hundreds of data streams.**

***Index Terms*—replicated, virtual log, stream storage, log structured, durability, ordering**

I. Introduction

Increasingly larger volumes of data are collected and stored in real-time from an increasingly higher number of sources (e.g., IoT devices, Cloud applications, etc.). They form distributed data streams that are essential for a large class of applications, ranging from real-time analytics to continual training and inference of AI models. This transition to real-time analytics was made possible in the past decade by advances in DRAM, with cheaper and larger memories available to Cloud computing. As such, processing systems such as Apache Spark [1] and Flink [2] have seen increasing adoption. While these runtimes enable efficient and fault-tolerant processing pipelines, the ingestion of data is a critical aspect upon which the performance of the entire system depends. This aspect is challenging, because it is not enough to provide access to the data streams with low-latency and high-throughput, but also to guarantee specific properties, such as durable acquisition that maintains the original order.

To this end, approaches such as Apache Kafka [3] and Pulsar [4], have gained significant traction. Such approaches enable data streams to be managed durably (i.e., data streams are replicated and persistently stored for fault-tolerance reasons) and consistently (i.e., preserving the order of appended stream records for efficient consuming and correct application semantics) by using log-structured storage [5]. However, with increasing number of sources and data volume, it is becoming increasingly difficult to achieve all desired properties simultaneously: durability, consistency, low latency, high throughput, scalability. This paper focuses on the aforementioned problem.

We consider the following ***stream storage model***. Streams follow a producer-consumer model, where N producers append events in parallel to M independent stream topics. To parallelize access to streams both for producers and consumers, stream storage systems partition topics. Producers can append new events atomically into a partition (therefore, storage systems acquire stream events in order according to arrival timestamp), but consumers can read at any offset (similarly to POSIX files, they can jump at any file offset and initiate sequential reads). Storage systems avoid losing events in case of failures through data replication. They replicate stream partitions using a form of eventual consistency: an append finishes after being successfully applied on all replicas. However, to avoid blocking the consumers during appends, new appends are not exposed immediately. Instead, the consumers are allowed to read events up to a given timestamp (the "tail"), incremented at the discretion of the runtime, which will eventually include all finished appends.

Our goal is to maximize the overall throughput of appends (with and without concurrent reads), given that multiple stream partitions get accessed in parallel. State-of-art stream storage systems (Kafka) implement replication of appends for each partition individually by using a replicated log. For performance reasons, new events are not sent separately to each log replica at fine granularity but instead, the runtime batches unreplicated stream events together. However, at scale, this still has a significant overhead due to contention (concurrent replication of appends on many different partitions) and resource utilization (too many headers and indices, etc.). Our challenge is then ***how to organize stream partitions for faster replication and better throughput*** compared to using one log per partition approach?

Towards this goal and in order to efficiently increase the number of managed streams (in terms of latency and throughput) by stream storage while reducing the resources required for ensuring data is not lost, this paper introduces the virtual

log design principle. We propose to separate data stream partitioning from log-based stream replication through a new technique called shared replicated virtual logs. The virtual log is responsible for consistently replicating multiple data streams while preserving the stream data order of associated data streams. Data streams get partitioned into multiple partitions that are further logically associated (for replication) with virtual logs by the storage system transparently to users.

We make the following contributions:

- We describe challenges introduced by state-of-art stream storage implementations.
- We introduce the virtual log design principle that explores the idea of separating partitioning and replication for stream storage design.
- We implement the virtual log technique on the KerA [1] ingestion system. We describe the original KerA partitioning mechanism (no support for replication) and then we describe the virtual log replication architecture and implementation.
- We evaluate the replicated KerA (with virtual logs) and the popular stream storage solution Apache Kafka. We demonstrate that the virtual log brings a trade-off between replication performance (throughput), replication capacity/memory resources (number of virtual logs) and the number of streams, while considering concurrent producers and consumers.
- We then review state-of-art stream partitioning techniques and how existing systems leverage log-structured storage to ensure durability through replication, we discuss future work and then conclude.

II. Motivation: Current Stream Storage Architectural Implementation

A. Background

Replication [6] is the standard solution used for ensuring fault-tolerant stream data storage. Through replication, the storage system stores multiple copies of stream data over multiple nodes called replicas. To maintain consistency among replicas, stream storage systems usually rely on a primary-backup mechanism that dedicates one primary replica for serving clients while backup replicas synchronizes with the primary replica through pull-based or push-based approaches.

A scalable stream storage system needs to accommodate multiple tenants efficiently, each pushing numerous data streams with different requirements for ingestion throughput and read/write access latency: for instance, it should efficiently support the ingestion of tens of massive streams (i.e., having tens of thousands of partitions) or the ingestion of millions of tiny streams (i.e., each having a few partitions). A stream is logically partitioned into multiple partitions, each partition being implemented with multiple logs, ensuring parallel writing and reading. Partitioning is thus a technique used for achieving scalability.

[1] KerA's source code is available at: https://gitlab.inria.fr/KerA/kera.

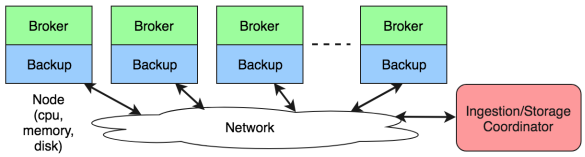

Fig. 1: The architecture of a stream storage system: the coordinator manages storage nodes on which live broker and backup processes. Clients mainly interact with brokers since they control the active partition replicas. Backups handle the passive partition replicas and are later involved in fault tolerance recovery.

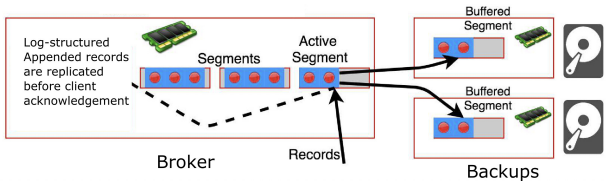

Fig. 2:

A partition is represented by a replicated log. Each new record of a stream's partition is appended to the log and is synchronously replicated to backups, after which client writes are acknowledged.

A Stream Storage Architecture. As represented in Figure 1, a stream storage architecture contains a single layer of B brokers that serve producers and consumers. On each node live, 1) one broker service with an ingestion component offering pub-sub interfaces to stream clients and 2) one backup service used for storing stream's replicas that are only read during crash recovery. A broker manages the main memory of a server and handles multiple streams by ingesting stream records into associated partition (by a record key). The broker handles requests (for data and stream metadata) from P producers and C consumers through (a)synchronous RPCs. Brokers and clients operate multiple streams S configured with a replication factor R that ensures each stream is replicated R times for durability purposes. To minimize the number of data copies, clients and brokers share a binary data format that allows data to be appended to brokers' data structures (from producers) or traversed (by consumers) without additional copies. Producers wait for the brokers and backups to acknowledge replicated data streams and eventually re-transmit data in case of errors.

Log-structured storage implementation. In stream storage systems, e.g., in Apache Kafka, stream partitions can be implemented by independently replicated ordered logs to easily and efficiently ensure durability and ordering. Each stream is partitioned into a fixed number of partitions P_s, each partition being backed by one replicated log L. Each replicated log is divided into segments, always one active segment on the broker accepting new appends, and replicated segments on the backups storing copies. Leveraging log-structured storage is a decision that comes naturally, being emphasized by the structure of a stream: data arrives in a record-by-record fashion, and it is processed and stored similarly (see Figure 2). This architectural choice enables simplified development

trading off cluster size with performance and the number of streams optimally managed.

B. Problem Statement

Writing Chunks From Producers to Brokers and Backups. Let us give an example of how stream data flows from producers to brokers and backups. One producer writes to one stream that has one partition managed by a cluster of three brokers and backups. One broker (called primary or leader) is responsible for the partition's active replica while the other two backups are used to hold two copies (passive replicas) of this partition. The partition is represented by a replicated log that holds an in-memory buffer on broker and on each of the backups replicating its data. The producer aggregates in each chunk a set of records and sends a few chunks (further grouped in a request) to the primary broker; once the primary broker acquires a chunk by appending it to the partition log, it also sends (replicates) the chunk in parallel to the backups. Once backups hold the acquired chunk, they respond immediately to the broker that finally acknowledges the producer's request. To ensure durability, backups asynchronously write buffered chunks to secondary storage. Therefore, the producer request is not impacted by secondary storage latency. The end to end latency of an appended chunk largely depends on the network transfer costs, while the broker and backups appends costs are minimized. The producer internally appends records to next chunk to be submitted to a broker, and waits up to a certain timeout before the chunk is sent in a new request. In practice, the chunk size, the request size, the timeout and the number of parallel producer requests are chosen such that the latency is minimized under a certain threshold while maximizing the throughput.

Chunk Replication Challenges. Replicating each producer chunk individually would dramatically reduce the number of streams that could actively, efficiently and durably ingest data, while reducing the overall stream storage ingestion throughput. Since each stream is implemented with tens of replicated logs (corresponding to tens of partitions), for large stream use cases (e.g., Netflix, Linkedin), a durable stream storage would have to manage tens of millions of small IOs which is not efficient. As such, we need to find a way to aggregate the producers' chunks before replicating them in order to: (1) increase the number of active streams the storage system can efficiently support and (2) improve the ingestion throughput (for small streams) by replacing small IOs with larger ones on backups. Towards these goals, we next introduce the virtual log shared replication principle.

III. The Virtual Log Design Principle

Building on the previous observations, we explore the idea of separating the partitioning and the replication in the design of stream storage systems.

We propose that streams organized into multiple partitions to be further logically associated with one or multiple ***virtual logs***, that are shared and replicated. Virtual logs are used to organize and replicate chunks of records acquired continuously from stream producers.

A virtual log resembles a shared replicated log (as represented in Figure 2) and is designed as follows. On the broker side, each virtual log is composed of an increasing number of ordered ***virtual segments*** that keep references to chunks that are physically part of multiple streams' partitions. Therefore, the virtual segment (implemented as an append-only in-memory buffer) holds the chunks' metadata it further uses to replicate the actual chunks to backups. Only one virtual segment is open to appends, while the closed ones are immutable. When a new virtual segment is opened, a set of distinct backups is chosen (potentially different from the ones associated to the previous virtual segment) for replicating in order its associated chunks. Distributing data to all backups helps at recovery time since data can be read in parallel from many backups. For durability, data is asynchronously written to secondary storage with the same in-memory format.

Multiple streams' partitions are associated with multiple virtual logs. With this approach, replicated virtual logs consolidate multiple replication RPCs by replacing small I/Os with larger ones on backups.

Note that stream partitions and virtual logs are distinct entities as they have different design goals and are supported by different data structures. While the stream partitions ensure correct ordering of stream records, virtual logs target: *(i)* fast replication, *(ii)* higher throughput and *(iii)* durability.

The replication throughput of a single stream can be customized by allowing the system/users to tune the ***replication capacity***, i.e., how many replicated virtual logs can be created for a single stream and how they are shared with other streams. Additionally, virtual logs reduce the number of write RPCs used for replication compared to using one replicated log per partition, increasing the ingestion throughput and the number of managed streams. The storage system can further increase the ingestion throughput of a high-throughput stream by associating more virtual logs. For durability (data is never lost in case of failures), each virtual log can be recovered in parallel over many brokers that become the primary leader of the partitions associated to recovered virtual logs.

Compared to state-of-art stream storage system implementations, we contribute with a consolidated approach that creates a single replicated log for multiple stream partitions. The virtual log combines events (grouped in chunks) from different partitions in the same batch, thereby increasing append throughput, while at the same time reducing the extra indexing overhead, thereby decreasing the resource utilization.

In the next section, after presenting a background on KerA's dynamic partitioning mechanism (KerA as described in [7] does not implement replication), we describe the implementation of the virtual log replication in KerA.

IV. Virtual Log Implementation in a High Performance Stream Storage System

KerA [7] is a high-performance ingestion system that unifies ingestion and storage, exposing one API that captures the

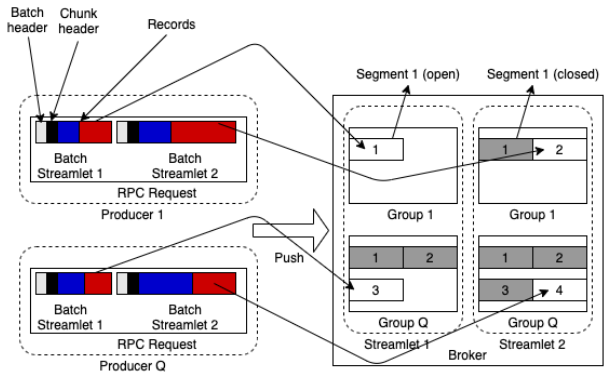

Fig. 3: Representation of records, chunks, segments, groups, and streamlets illustrates how producers aggregate records into requests and push them to brokers. Each request contains multiple chunks (chunk and request size are configurable). We exemplify a stream composed of two streamlets. A streamlet holds up to Q active groups (fixed-size sub-partitions) for new appends. A producer writes to the streamlet's active group corresponding to the entry calculated as producer identifier modulo Q.

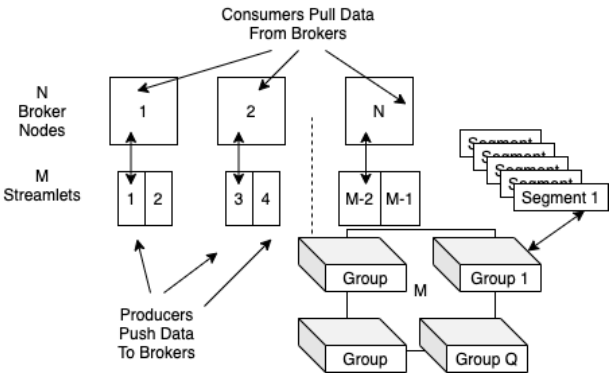

Fig. 4: Semantic partitioning of streams with streamlets. A stream is composed of a set of streamlets (partitions). Each streamlet is composed of an unbounded set of fixed-size sub-partitions called groups of segments.

semantics of both stream-based systems like Apache Kafka and distributed systems like Hadoop HDFS. To implement scalable data ingestion, KerA proposes two core ideas: (1) dynamic partitioning based on semantic grouping and sub-partitioning, which enables more flexible and elastic management of stream partitions; (2) lightweight offset indexing (i.e., reduced stream offset management overhead) optimized for sequential record access. KerA builds atop RAMCloud's [8] RPC framework to leverage its network abstraction that enables the use of other network transports (e.g., UDP, DPDK, Infiniband), thus offering an efficient RPC framework on top of which we develop KerA's client RPCs. Moreover, this allows KerA to benefit from a set of design principles like ***polling and request dispatching*** [9] that help boost performance (e.g., kernel bypass, and zero-copy networking are possible with DPDK and Infiniband). KerA exposes stream-based APIs in C++ and Java and integrates with Apache Flink.

A. Dynamic Stream Partitioning in KerA

Let us introduce the main concepts KerA leverages for the ingestion and storage of data streams. In addition to the data partitioning model presented in [7], we add support for chunks of records at client and broker levels. The virtual log further leverages chunks for consolidating data replication.

A ***stream*** is an unbounded sequence of records that are not necessarily correlated with each other. An ***object*** is simply represented as a bounded stream. Figure 3 illustrates the conceptual representation of records, chunks, segments, groups and streamlets. Each ***record*** of a stream is represented by an entry header which has a checksum covering everything but this field; the record is defined by several keys (possibly none) and its value, similar to the multi-key-value data format used in RAMCloud [10]. The record's entry header contains an attribute to optionally define a version and a timestamp field that are necessary to enable key-value interfaces efficiently. Record entries are further grouped by producers into ***chunks***, each chunk having a configurable fixed size (e.g., 16 KB). The chunk aggregation is useful for two reasons. First, it gives clients the chance to efficiently (for metadata purposes) batch more records in a request in order to trade-off latency and throughput. Second, since each chunk is tagged with the producer identifier and with a dynamically assigned partition offset identifier (based on streamlet group), this helps ensuring exactly once semantics and ordering semantics necessary for durable ingestion and consistent processing.

A producer client prepares and writes a request containing a set of chunks. Each chunk acquired by the storage system is appended into a ***segment*** represented by an in-memory buffer managed by the broker. A segment has a customizable fixed size (e.g., 8 MB) necessary for efficiently moving data from memory to disk and backwards (on brokers, processes handling data access, segments have the same structure on both disk and memory). To reduce the metadata necessary to describe the unbounded set of segments of a stream, we further logically assemble a configurable number of segments into a ***group***. In this way, each stream can be logically represented by a smaller, unbounded set of groups of segments. Each group is further assigned to one consumer/producer to load balance data (for higher parallelism) and increase processing/ingestion throughput.

A stream is composed of logical partitions called streamlets that contain ordered stream records by a key (Figure 4). To increase write and read parallelism, a ***streamlet*** is further divided into fixed-size sub-partitions (groups of segments), with each group created dynamically as data arrives (active groups are created as needed while closed groups suffer no appends). A stream has up to M number of streamlets initially created on a set of N, $N \leq M$, number of brokers. M represents the maximum number of nodes that can ingest and store a stream's records (ensuring horizontal scalability through migration of streamlets to new brokers). Each streamlet can contain an unlimited number of groups that can be processed in parallel by multiple consumers (ensuring vertical scalability), of which up to Q active groups correspond to physical sub-partitions that allow parallel appends from numerous producers.

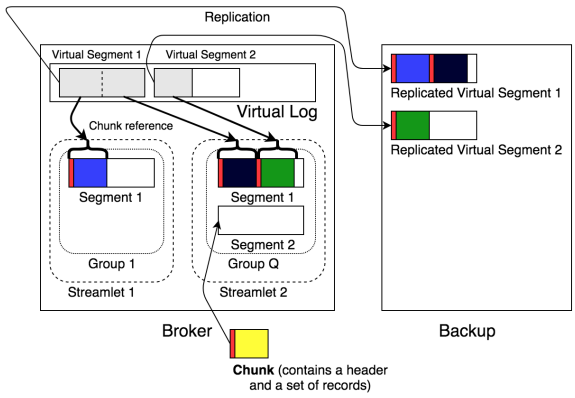

Fig. 5: The virtual log technique for adaptive and fine-grained replication. In this example a stream with 2 streamlets is created and managed by one broker, with one backup used to replicate the stream. We create and maintain in-memory a log-based data structure to represent our replicated virtual log that aggregates and replicates chunks of different groups of streamlets of a single stream.

B. Adaptive and Fine-Grained Replication for Multiple Streams: The Replicated Virtual Log

This section describes the virtual log replication design and implementation in KerA, suitable for the efficient and durable ingestion of multiple streams.

To maximize the number of streams the storage system can manage at a given time, we associate with a stream a set of ***virtual logs***, each virtual log being composed of replicated virtual segments. Each virtual segment contains ***references*** to chunks physically stored on stream partitions (associated with their creation in order). The virtual segment's chunks are replicated into a corresponding backup in-memory segment. The backup asynchronously writes the segment on storage to ensure durability. The backup's segments contain chunks from possibly various groups of different streamlets of multiple streams. Backups read segments from disk and issue writes to the new brokers responsible for recovering a crashed broker's lost data at recovery time. Each of these requests is handled as a normal producer request (i.e., chunks are ingested into their respective groups) while metadata is safely reconstructed.

We give an example in Figure 5 where a virtual log composed of two virtual segments replicates one stream as follows: the virtual segment 1 does not accept new appends since it is closed and fully replicated on the backup (see replicated virtual segment 1); the virtual segment 2 is open, accepts new appends and has a reference to a chunk (see pointer to segment 1 of the group Q under streamlet 2) with its corresponding data already replicated on the backup (see replicated virtual segment 2). Assuming we have a new chunk of data to be appended to the group Q of streamlet 2, since segment 1 has no remaining space for this chunk, a new segment 2 gets created and the chunk physically appended to it. Then, a reference to this chunk is appended to the virtual log: since the virtual segment 2 is open and has enough space (virtually) to hold this chunk, the chunk reference is saved. Perhaps, in parallel, there are new chunks in segment 2; one of the subsequent write requests will trigger the replication of the available chunks and store them on backup's replicated segment 2. The virtual segment only keeps chunk metadata and calculates its remaining virtual space based on the accumulated chunk lengths.

Each virtual log is composed of a set of virtual segments to be replicated, always a single open virtual segment (the replication of the virtual log resembles the RAMCloud's log implementation). Each virtual segment contains references to the chunks appended to the streamlets' physical segments. The virtual log stores the chunk metadata and keeps a reference to the physical segment containing the chunk's records. The virtual segment has a header with a checksum that covers the chunk's checksums. Backups use this information for recovery and data integrity. The virtual segment also keeps two attributes: one to denote the next available/free offset (the header) and another that points to what was already durably replicated (the durable header) – similar attributes are kept for each physical segment. The virtual segment keeps an ordered list of references to chunks; the chunk metadata contains a reference to the physical segment and the chunk's offset into physical segment and length. The replication implementation ensures that each chunk is atomically replicated thus, the durable header points to the next chunk to be replicated. After a chunk is replicated, the runtime updates the durable head of the physical segment so that consumers can pull records up to it.

Replicating chunks after broker appends. Each producer request is characterized by the stream and producer identifiers and a set of chunks, each one characterized by its length and a streamlet identifier. The broker identifies the stream object corresponding to the stream identifier and then, for each chunk, identifies the streamlet active group based on the producer identifier and parameter Q (how many active groups a streamlet is configured with). The chunk is appended to the active group (this operation internally creates a new segment and/or a new group if necessary), and then a chunk reference is appended to the replicated virtual log corresponding to the streamlet of the active group entry. Once all chunks of a request are appended, the corresponding replicated virtual logs are synchronized on backups to replicate its chunks.

Building producer requests: management of chunks. Figure 3 illustrates how producers organize stream records into requests that contain multiple chunks. Being able to send more chunks in a request helps mitigate latency and throughput trade-off. A chunk is appended to the open segment of the corresponding active group (the group is computed at the broker side based on producer and streamlet identifiers found in the request header). The broker only uses the portion of the request starting with the chunk header (including subsequent records) and appends it to a group's open segment. Each append operation can lead to creating a new segment or a new group. The chunk header (reference) contains attributes for the corresponding ***[group, segment]*** of a streamlet that are updated at append time: these attributes are essential at recovery time to consistently reconstruct each group (additionally, each

segment is tagged with the stream and streamlet identifiers).

Each producer implements two threads that communicate through shared memory (a set of empty chunks are reused for the next requests). For each new stream record the first thread identifies an available chunk where the record is appended according to the partitioning strategy (round-robin or by record's key, which is hashed to identify a streamlet). For each streamlet, a set of chunks are dynamically created in the shared memory. The second thread creates the next requests: it gathers a set of filled chunks up to the request size, one for each streamlet, and manages the RPC invocations.

V. Results

A. Setup and Parameter configuration

We run all our experiments on Grid5000 [11] on the ***Paravance*** cluster with each server characterized by 16 cores and 128 GB of memory. A single layer of brokers serves both producers and consumers. In our experiments, we install on each server a broker service (the broker implements an ingestion component offering RPC interfaces to streaming producer and consumer clients) and a backup service used for storing stream's replicas. This configuration with nodes serving both clients and internal replicas is similar to production environments. We configure a broker with 16 threads that correspond to the number of cores of a node; a broker holds one copy of the stream records. In each experiment, we run an equal number of concurrent producers and consumers. Each client is associated with at least one stream or partition. When replication is enabled, we configure one or more virtual logs that manage the stream replication while consumers only pull durably replicated data. A producer/consumer request is characterized by its size (i.e.., ***request.size***, in bytes) and contains a set of chunks, one for each partition, each chunk having a ***chunk.size*** in kilobytes.

In each experiment, the source thread of each producer (as illustrated in Figure 6) creates up to 100 million non-keyed records of 100 bytes and partitions them in chunks of configurable size, one stream/partition at a time. We use synthetic data similar to the open messaging stream benchmark [12]. The source waits no more than 1 ms (parameter named ***linger.ms*** in Kafka) for a chunk to be filled. After this timeout, the producer marks the chunk ready to be sent to the corresponding broker. Another producer thread aggregates chunks in requests (one request for each broker) and sends them to the broker (one synchronous TCP request per broker, multiple parallel requests). Similarly (as illustrated in Figure 7), each consumer pulls chunks of records with one thread and iterates over serialized records on another thread. Each client has a cache of up to 1000 chunks. In the client's main thread, we measure the ingestion and processing throughput (e.g., million records per second) and log it after each second. Producers and consumers run on different nodes. We compute the average ingestion/processing throughput per cluster for up to 60 seconds worth of each experiment, measured while concurrently running all producers and consumers (without considering each client's first few seconds measurements until all clients have similar workload).

Producers are configured as ***proxy clients*** and share the streams: they compete when producing stream records (since each request contains a chunk for each stream partition). When the number of active groups per streamlet is higher than 1, appends on brokers can happen in parallel. Proxy-based producers represent the worst-case scenario, and most streaming use cases are implemented similarly (each RPC request contains one chunk for each stream partition to target similar record latencies). Another option is to configure one producer for each stream: this brings no competition between producers and can be used by certain HPC simulations where each simulation core is configured to produce data partitioned by its number. We choose the first option since it represents real-world scenarios (e.g., streaming sink operators are similar to proxy-based producers) while it brings more pressure on broker implementation (e.g., due to competition between RPCs, concurrency between appends). Moreover, producers aggregate chunks in a single request to reduce the communication overhead due to TCP, while reducing latency processing of records that are part of different streams living on the same broker. Thus, we batch more chunks in a request, up to a certain size that we evaluate in various experiments. Consumers are configured to pull data exclusively from their associated streams. Consumers also pull more chunks in a request if there are more streams on a broker.

In the following experiments, we configure a fixed number of four nodes that serve as brokers and backups. At the same time, we increase the number of clients (concurrent producers and consumers), the number of streams (and their partitions), the chunk size (batching more records per chunk), the replication factor (how many copies the system manages), and the number of virtual logs used for replication.

B. Replicated KerA versus Kafka

This section compares the cluster throughput between Apache Kafka and KerA while considering a configuration with four brokers on the Grid5000 Paravance cluster (16 cores per broker). Apache Kafka and KerA have a similar architecture (coordinator, brokers), but they handle partition and replication differently. Each stream (called a topic in Kafka) gets partitioned in a fixed number of partitions, each partition being backed by one replicated log. One Kafka broker is the partition leader (active replica) responding to clients' requests, while other Kafka brokers are assigned as partition followers (passive replicas), issuing pull-based requests to fetch the latest partition data (passive replication). In KerA, a stream can be partitioned in a fixed number of partitions called streamlets, while each streamlet is further partitioned in fixed sub-partitions created dynamically. KerA associates shared replicated logs (called virtual logs) with streamlets, separating the replication implementation from partitioning. A KerA broker is the partition leader, while other KerA backups synchronously replicate data once the leader acknowledges each append (active replication). One has to tune the Kafka

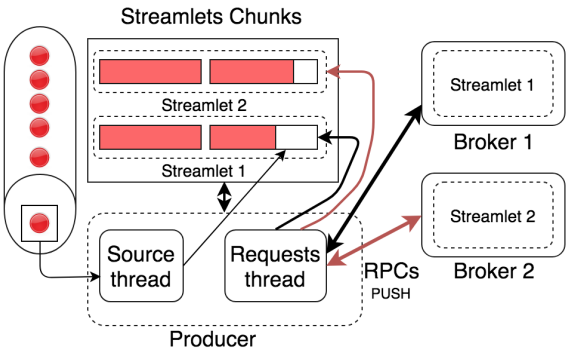

Fig. 6: Producer architecture. The Source thread appends records to in-memory chunk buffers. The Requests thread batches one chunk for each streamlet (if the chunk is filled or a configurable timeout per chunk passed) in requests and pushes them to brokers.

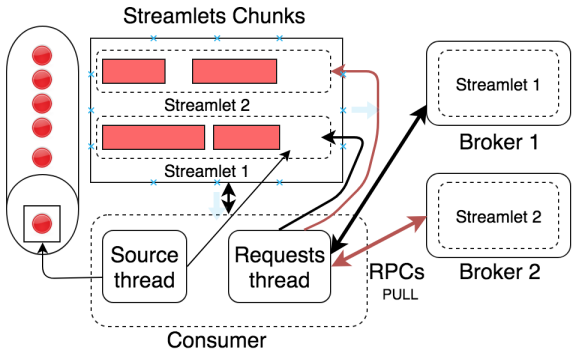

Fig. 7: Consumer architecture. The Requests thread builds one request for each broker and pulls one chunk for each streamlet associated to the consumer. The Source thread consumes in-order one chunk per streamlet: it iterates the chunk and creates records. The source could internally push these records to other streaming operators for further processing.

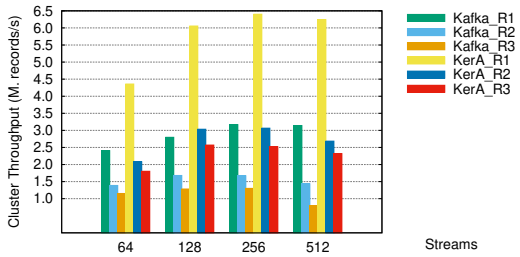

Fig. 8: Scaling the number of streams. Kafka versus KerA with 4 concurrrent producers writing over 4 brokers, chunk size 1 KB. In KerA each streamlet has one sub-partition, being configured like a partition in Kafka. A stream has one partition. KerA uses for replication four virtual logs per broker shared by all streams. Each producer request to one broker contains a chunk for every associated partition of that broker/request. R1, R2, and R3 correspond to replication factor one, two and respectively three.

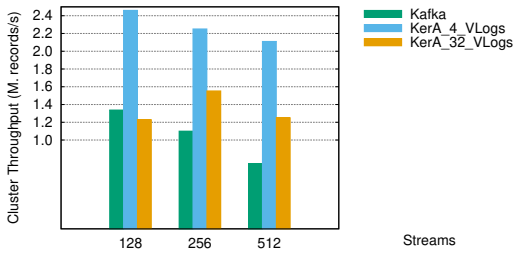

Fig. 10: Low-latency configuration. Kafka versus KerA while varying the number of streams, replication factor three with the chunk size of 1 KB. The Cluster throughput corresponds to 4 producers running in parallel with 4 consumers on 4 brokers. KerA is partitioned similarly to Kafka, one streamlet (partition) per stream with one active sub-partition per streamlet (no parallel appends). KerA has 2 configurations for stream replication, with 4 and respectively 32 virtual logs per broker.

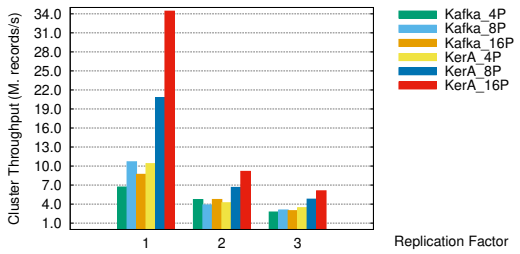

Fig. 9: Scaling the number of clients. Kafka versus KerA while increasing the replication factor, running concurrent producers with the chunk size of 16 KB. The Cluster throughput corresponds to 4, 8 and respectively 16 concurrent producers running on 4 brokers. KerA is configured similarly to Kafka, one replicated log per partition, 128 streams each having one partition.

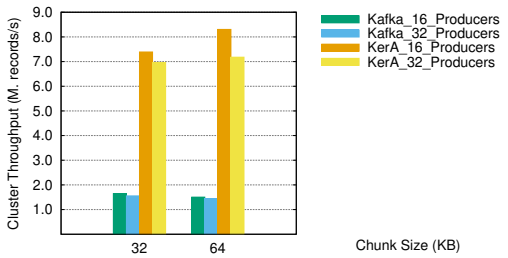

Fig. 11: High-throughput configuration. Kafka versus KerA while varying the number of producers and chunk size, with replication factor three over 4 brokers. A stream with 32 partitions is created in Kafka, while a stream with 32 streamlets is created in KerA, one streamlet has 4 active sub-partitions. KerA configures one virtual logs per sub-partition.

replication followers to be efficiently in sync with their leaders. Producers' appends get acknowledged once data has been replicated and owned by leaders and followers of a partition. In parallel, consumers can only process data that was durably acknowledged.

We run the first set of experiments with concurrent producers that write (in parallel) to all brokers and all streams/partitions. As illustrated in Figure 8, for a fixed chunk size of 1 KB, while increasing the number of streams, throughput increases (since more records are batched for each RPC). While increasing the replication factor from one to three, throughput decreases as expected. While in Kafka partition's replicas are kept in sync using one replicated log per partition, in KerA we leverage the virtual log mechanism dedicating

four virtual logs per broker, each shared virtual log batching multiple partition chunks for replication. This configuration helps increasing throughput for such configurations optimized for latency (less batching at the producers' side).

Next, as illustrated in Figure 9, we fix the number of streams to 128 (equally distributed over four brokers), the chunk size to 16 KB, and we increase the number of concurrent producers from 4 to 16 while also increasing the replication factor from one to three. Running more producers helps increase the total throughput for the configured cluster, but while increasing the replication factor, throughput decreases. For the configuration with 16 producers, KerA obtains 2x better throughput when the replication factor is three. Although configured similarly, KerA's active replication architecture (versus passive replication in Kafka) uses cluster resources more efficiently, while it requires no tuning. In Kafka, one has to tune the partition followers (the passive replicas trying to catch with the active replica) to pull enough data to remain in sync with leaders.

For the following experiments, we configure an equal number of consumers to run in parallel with producers. Producers and consumers run on different nodes. For a configuration suitable for the low-latency ingestion of hundreds of small streams, as illustrated in Figure 10, when configured similarly to Kafka, e.g., for the 128 streams configuration, Kafka and KerA have similar performance. Still, when reducing the number of virtual logs used for replication, KerA obtains up to 3x better performance. The virtual log setting introduces not only a good trade-off for reducing the resources used for fast replication of many small streams, but it can also improve performance by reducing the number of RPC requests, aggregating small replication RPCs, and thus reducing communication overhead. An architecture based on active replication like in KerA is more suitable for ensuring low-latency and high-throughput data ingestion and avoiding other parameters for tuning a passive replication mechanism as in Kafka. In addition, it gives more control for providing data is durably replicated.

We further experiment with Kafka with a configuration optimized for throughput. As shown in Figure 11, KerA obtains up to 5x better cluster throughput when the data replication factor is three. KerA benefits from its dynamic partitioning mechanism and the virtual log setup, allowing to associate one virtual log with replicating each sub-partition.

C. Impact of the Virtual Log on Performance when Optimizing for Latency

The following experiments have a configuration optimized for latency as follows. The chunk size is fixed to 1 KB while producers wait no more than one millisecond before pushing a chunk to the stream storage, or a filled chunk is immediately pushed. Consumers continuously pull data up to one chunk per stream/partition with no delay. We experiment with an increasing number of streams, each having one partition. Four consumers run in parallel with four producers and pull durably replicated data (replicated data is acknowledged synchronously by backups to brokers before responding to clients).

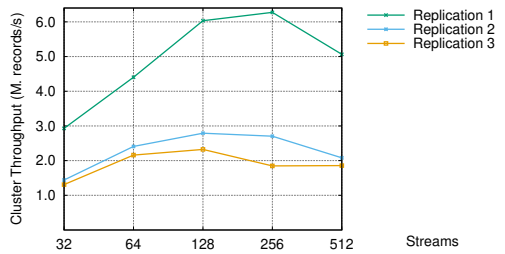

Fig. 12: Scaling the number of streams in KerA. One replicated virtual log per broker is shared for the replication of up to 512 streams. Replication 1, 2 and respectively 3 corresponds to one, two, and respectively three copies of data. 8 concurrent producers and consumers, 4 brokers, chunk size 1 KB.

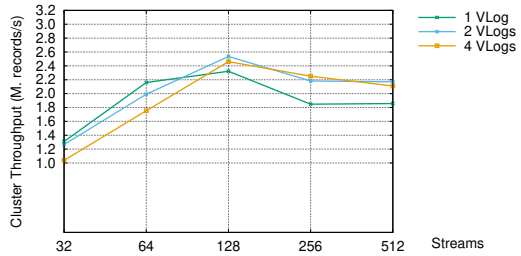

Fig. 13: Increasing the replication capacity (1, 2 and 4 shared replicated virtual logs per broker) while scaling the number of streams. Replication factor three, 8 concurrent producers and consumers, 4 brokers, chunk size 1 KB.

Our goal is to understand the impact of these configurations while increasing the replication factor (up to 3 copies, including the broker) and the number of virtual logs used for replication. Producers share all streams, building requests that contain up to one chunk for each stream if a chunk contains at least one record. Therefore, each producer submits a number of requests equal to the number of brokers over multi-TCP connections. In parallel, the same number of TCP requests is executed by each consumer, competing on cluster resources.

While state-of-art stream storage systems configure one log per stream, we experiment with an increasing number of shared logs by all streams of a broker through the virtual log implementation. Can we obtain better performance with a reduced number of replicated virtual logs? In the following experiments, the cluster throughput (measured in million records per second) corresponds to the aggregated throughput of four producers. Four consumers pull the same data concurrently to producers.

As illustrated in Figure 12, using one virtual log we can ingest up to 1.8 Million records/s for 512 streams with replication factor three (three copies of data), while using 4 brokers (64 cores). Configurations with two and four virtual logs (see Figure 13) help increase the cluster throughput with up to 30-40%.

However, when we further increase the number of virtual logs for the replication of 128, 256, and respectively 512 streams (see Figures 14, 15, and respectively 16), we observe that the cluster throughput performance can drop by up to 40-50%. The virtual log can save resources used to replicate hundreds of streams and it allows the stream storage to

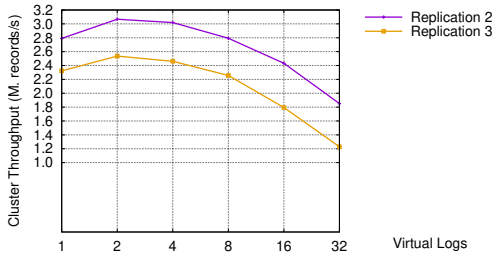

Fig. 14: Ingestion of 128 streams varying the number of virtual logs. 8 concurrent producers and consumers, 4 brokers, chunk size 1 KB.

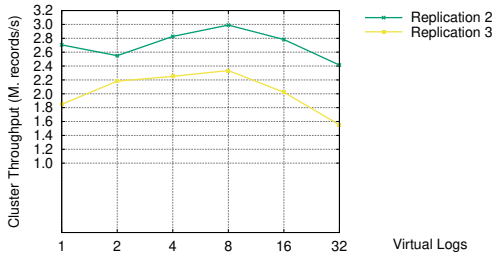

Fig. 15: Ingestion of 256 streams varying the number of virtual logs. 8 concurrent producers and consumers, 4 brokers, chunk size 1 KB.

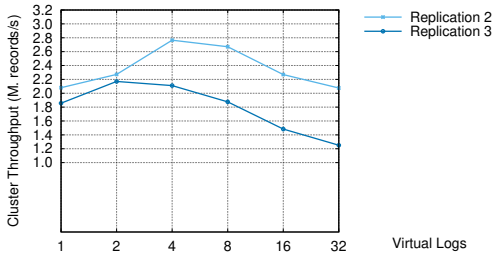

Fig. 16: Ingestion of 512 streams varying the number of virtual logs. 8 concurrent producers and consumers, 4 brokers, chunk size 1 KB.

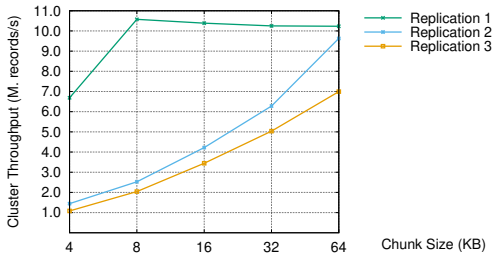

Fig. 17: One virtual log per sub-partition. The Cluster throughput corresponds to 4 producers running in parallel with 4 consumers. 4 brokers, 32 shared virtual logs per broker.

improve the ingestion performance when carefully configured.

D. Impact of the Virtual Log on Performance when Optimizing for Throughput

The following experiments have a configuration optimized for throughput as follows. We create one stream having 32 streamlets (partitions) on four brokers. Each broker is equally responsible for 8 streamlets, with each streamlet configured with 4 sub-partitions (called active groups in KerA) that allow for parallel appends on a sub-partition. We increase the chunk size from 4 KB to 64 KB while also increasing the number of producers (and consumers) from 4 to 32. Increasing the chunk size allows the producer to batch more records while increasing the number of clients puts more pressure on the stream storage system.

As illustrated in Figure 17, when we configure one virtual log for each sub-partition (32 virtual logs per broker), we obtain up to 7 Million records per second when increasing the chunk size up to 64 KB, while a total of 8 clients produce and consume records in parallel.

Let us further increase the number of clients hoping to increase the cluster throughput while keeping the configuration of one virtual log per sub-partition (similarly to Kafka). For applications that want both a high throughput and lower latency, keeping the chunk size smaller forces the system designer to increase the number of clients at the expense of higher competition between them when pushing data to brokers. Configurations with 16 and respectively 32 clients (see Figures 18 and respectively 19) obtain 8.3 million records/s when the chunk size is 64 KB and the replication factor is three. Further doubling the number of clients puts more pressure on the storage system (4 brokers configured with a total of 64 cores serving both clients and backup replicas). As shown in Figure 20, a configuration with 64 clients obtains up to 7.2 million records/s; although throughput reduces compared to the maximum of previous configurations, having more clients produce and consume data helps to reduce the processing latency.

We further fix the number of clients to 16, and we variate the number of virtual logs from 1 to 32 (see Figure 21). When the chunk size is 32 KB or 64 KB, the configurations with 8 and 16 virtual logs obtain a slightly higher (an extra of 300K records/s) throughput than the configuration with 32 virtual logs.

E. Discussion

KerA relies on dynamic partitioning, lightweight indexing, custom memory management, and virtual log techniques to improve ingestion and processing performance. For high-performance ***durable ingestion***, KerA internally associates shared replicated virtual logs to multiple streams. The virtual log design allows a trade-off among consistency (through sync or async APIs, replication parameters), durability (one or more replicas), and scalability (increasing the number of streams, tuning the virtual log count, trading memory for many streams and performance). Overall, we observe that introducing the virtual log (as a replication mechanism for replicating data streams) improves cluster ingestion performance for both latency and throughput optimized scenarios. In Kafka, users may have difficulties ingesting both low-latency and high-throughput data streams on the same cluster since partition followers have a static configuration for how much data to pull to remain in sync with partition primary: it is difficult to tune for both scenarios. In KerA, the system easily allows

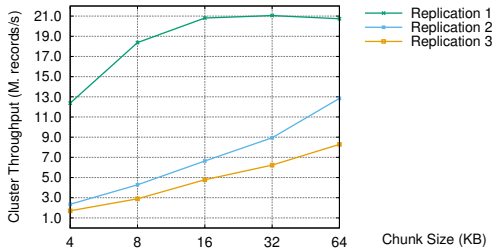

Fig. 18: One virtual log per sub-partition. The Cluster throughput corresponds to 8 producers running in parallel with 8 consumers. 4 brokers, 32 logs per broker.

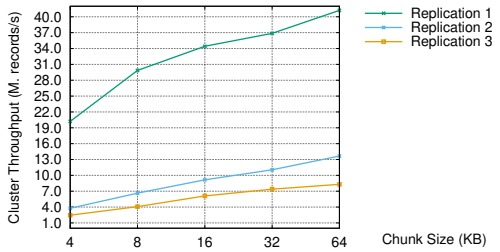

Fig. 19: One virtual log per sub-partition. The Cluster throughput corresponds to 16 producers running in parallel with 16 consumers. 4 brokers, 32 shared virtual logs per broker.

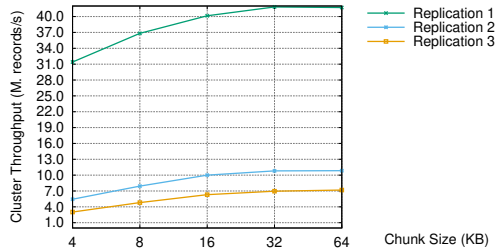

Fig. 20: One virtual log per sub-partition. The Cluster throughput corresponds to 32 producers running in parallel with 32 consumers. 4 brokers, 32 shared virtual logs per broker.

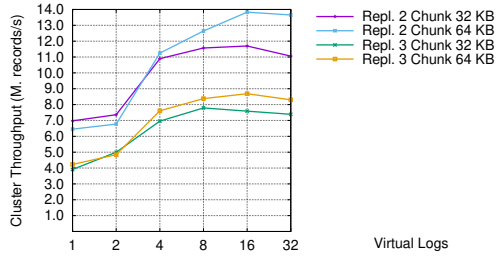

Fig. 21: Varying the number of virtual logs, replication with chunk size of 32 KB and 64 KB. The Cluster throughput corresponds to 8 producers running in parallel with 8 consumers. 4 brokers, one stream, 32 streamlets (partitions), each streamlet having 4 active groups (sub-partitions).

users to increase the ***replication capacity***, starting with one shared replicated shared virtual log per stream.

Future Work. KerA can benefit from the inherited RAMCloud's fast crash recovery implementation [13], harnessing the power of multiple servers to recover in parallel a crashed broker. Towards an extensive evaluation of crash recovery in KerA, we are looking to study co-located storage and stream processing architectures and leverage RDMA through the data flow interface [14]. KerA brings minimum overhead over RAMCloud's RPC mechanism while borrowing the dispatch-worker threading mechanism for handling RPCs; from initial measurements we expect the latency of small producer chunks to be similar to RAMCloud's measurements (tens to hundreds microseconds). In order for very high-throughput streams to not interfere with low-latency and high-throughput streams, we believe mechanisms like core-aware scheduling [15] can help. Optimizing latency in the normal case as well as when handling crash recovery is an important open problem for stream storage and processing architectures.

VI. Related Work

State-of-art stream storage systems, e.g., Apache Kafka [3], Apache Pulsar [4] or Distributedlog, [16], employ a ***static partitioning*** scheme where the stream gets split among a fixed number of partitions, each of which is unbounded, ordered, immutable sequence of records continuously appended. Each broker is responsible for one or multiple partitions, with each partition represented by a replicated log, e.g., as in Figure 2. Producers accumulate records in fixed-sized batches, each one appended to its corresponding partition. To reduce communication overhead, the producers will group a set of batches in a request per broker. The storage system ensures consumers only pull data for processing up to a log head that corresponds to durably replicated data.

Stream storage systems like Pravega [17], or KerA [7] employ a ***dynamic partitioning*** scheme where the stream is configured initially with a fixed number of partitions (named segments in Pravega or streamlets in KerA). While in Pravega, the segment can be split into other segments, or a few segments can be merged back to one segment, in KerA, the streamlet is split into fixed-size groups of segments created as data arrives. The KerA streamlet ensures multiple producers can append data in parallel while consumers can scale up and down as needed based on the number of groups.

To ensure durability, KerA relies on the log-structured methods developed in the RAMCloud project [8], adding support for handling multiple replicated logs. The streamlet concept backed by a multi-log (i.e., one replicated log for each active group of a streamlet) was implemented in the framework of the Ph.D. thesis of Yacine Taleb [18], applying RDMA techniques like in Tailwind [19]. However, although this option simplifies the implementation of crash recovery (relying on fault-tolerant techniques implemented in RAMCloud), for use cases where a large number of streams is needed, this technique needs higher in-memory storage support.

In the framework of the Ph.D. thesis of Ovidiu Marcu that describes KerA [20] (available Jan. 2019), we preliminarily introduced the virtual log concept for replicating data streams. The virtual log separates storage replication (used for ensuring durability) from data ingestion (used for providing ordering and partitioning). The virtual log ensures

consistency through chunks' metadata that are replicated along with stream records. Later in 2020, Delos [21] similarly used the virtual log abstraction to support different underlying log implementations. Compared to vCorfu [22] (that multiplexes homogeneous streams over a shared log), Delos uses the shared log abstraction for heterogeneous log implementations. Delos's virtual log exposes an API to clients that are not aware of underlying loglets implementations. Therefore, Delos can switch the underlying log implementation dynamically, leveraging other log protocols. KerA's clients use the Streamlet APIs (reads, writes), unaware of the virtual log. Compared to KerA's virtual log, vCorfu uses the shared log abstraction to provide a total order when writing to multiple streams, while log an stream replicas are handled separately, exposing vCorfu's clients to both stream and log APIs.

VII. Conclusion

We have proposed the virtual log-structured storage technique for efficiently handling the durable ingestion of multiple streams accessed in parallel by multiple producers and consumers. The virtual log enables zero-copy, adaptive, fast and fine-grained replication that allows the storage system to save processing resources when handling hundreds of streams. We argue that the virtual log-structured support is essential for Big Data analytics as many real-life applications (e.g., Netflix, Twitter, Facebook, Linkedin) require the management of a large number of streams with high performance. Experimental evaluations show that KerA outperforms Kafka, improving the cluster throughput when multiple producers write over hundreds of data streams, by up to 2x for latency-optimized producers and by up to 4x for throughput-optimized producers (ingestion configured with replication factor three). Our next steps are the evaluation of pipelined producers and consumers for various HPC and Big Data real-time use cases. Furthermore, through virtual logs, we can explore other stream partitioning mechanisms that enable non-sequential data accesses or easily integrate key-value stores based on log-structured storage (e.g., RocksDB).

Acknowledgment

This work was partially funded by ANR OverFlow (ANR-15-CE25-0003) and by BigStorage, a project funded by the European Union under the Marie Sklodowska-Curie Actions (H2020-MSCA-ITN-2014-642963). The experiments presented were carried out on the Grid'5000 testbed [23]. We thank anonymous reviewers for their feedback improving this paper.

References

[1] "Apache Spark," 2018. [Online]. Available: https://spark.apache.org/

[2] "Apache Flink," 2018. [Online]. Available: https://flink.apache.org/

[3] "Apache Kafka," 2018. [Online]. Available: https://kafka.apache.org/

[4] "Apache Pulsar," 2018. [Online]. Available: https://pulsar.apache.org/

[6] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso, "Understanding replication in databases and distributed systems," in *Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000)*, ser. ICDCS '00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 464–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850927.851782

[5] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, p. 1–15, Sep. 1991. [Online]. Available: https://doi.org/10.1145/121133.121137

[7] O. Marcu, A. Costan, G. Antoniu, M. Pérez-Hernández, B. Nicolae, R. Tudoran, and S. Bortoli, "Kera: Scalable data ingestion for stream processing," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 1480–1485. [Online]. Available: https://hal.inria.fr/hal-01773799/file/ICDCS_2018_paper_732.pdf

[8] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015. [Online]. Available: http://doi.acm.org/10.1145/2806887

[9] C. Kulkarni, A. Kesavan, R. Ricci, and R. Stutsman, "Beyond Simple Request Processing with RAMCloud," *IEEE Data Eng.*, 2017.

[10] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "Slik: Scalable low-latency indexes for a key-value store," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16. Berkeley, CA, USA: USENIX Association, 2016, pp. 57–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3026966

[11] "Grid5000," 2018. [Online]. Available: https://www.grid5000.fr/

[12] "The openmessaging benchmark framework." [Online]. Available: http://openmessaging.cloud/docs/benchmarks/

[13] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast Crash Recovery in RAMCloud," in *23rd SOSP*. ACM, 2011, pp. 29–41.

[14] L. Thostrup, J. Skrzypczak, M. Jasny, T. Ziegler, and C. Binnig, "Dfi: The data flow interface for high-speed networks," in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD/PODS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1825–1837. [Online]. Available: https://doi.org/10.1145/3448016.3452816

[15] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/qin

[16] G. Sijie, D. Robin, and S. Leigh, "Distributedlog: A high performance replicated log service," in *IEEE 33rd International Conference on Data Engineering*, ser. ICDE'17. IEEE, 2017. [Online]. Available: http://ieeexplore.ieee.org/document/7930058/

[17] "Pravega," http://pravega.io/.

[18] Y. Taleb, "Optimizing Distributed In-memory Storage Systems: Fault-tolerance, Performance, Energy Efficiency," Theses, ENS Rennes, Oct. 2018. [Online]. Available: https://hal.inria.fr/tel-01891897

[19] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes, "Tailwind: Fast and Atomic RDMA-based Replication," in *ATC '18 - USENIX Annual Technical Conference*, Boston, United States, Jul. 2018, pp. 1–13. [Online]. Available: https://hal.inria.fr/hal-01676502

[20] O.-C. Marcu, "KerA: A Unified Ingestion and Storage System for Scalable Big Data Processing," Theses, INSA Rennes, Dec. 2018. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01972280

[21] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, J. Liu, F. Gruszczynski, X. Zhang, H. Hoang, A. Yossef, F. Richard, and Y. J. Song, "Virtual consensus in delos," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 617–632. [Online]. Available: https://www.usenix.org/conference/osdi20/presentation/balakrishnan

[22] M. Wei, A. Tai, C. J. Rossbach, I. Abraham, M. Munshed, M. Dhawan, J. Stabile, U. Wieder, S. Fritchie, S. Swanson, M. J. Freedman, and D. Malkhi, "vcorfu: A cloud-scale object store on a shared log," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 35–49. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/wei-michael

[23] R. Bolze *et al.*, "Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed," *International Journal of High Performance Computing Applications*, pp. 481–494, 2006.