Chimera: An Elastically Scalable High Performance Streaming System with a Shared Nothing Architecture

Pascal Lau, Paolo Maresca

Infrastructure Engineering, TSG
Verisign
1752 Villars-sur-Glâne, Switzerland
Email: {plau, pmaresca}@verisign.com

Abstract—**On a daily basis, Internet services experience growing amount of traffic that needs to be ingested first, and processed subsequently. Technologies to streamline data target horizontal distribution as design tenet, giving off maintainability and operational friendliness. The advent of the Internet of Things (IoT) and the progressive adoption of IPv6 require a new generation of scalable data streamline platforms, bearing in mind easy distribution, maintainability and deployment. Chimera is an ultra-fast and scalable Extract Transform and Load (ETL) platform, designed for distribution on commodity hardware, and to serve ultra-high volumes of inbound data, processing in real-time while offering a simplistic fault model. It strives at putting together top performance technologies to solve the problem of ingesting huge amount of data delivered by geographically distributed agents. It has been conceived to propose a novel paradigm of distribution, leveraging a shared nothing architecture, easy to elastically scale and to maintain. It reliably ingests and processes huge volumes of data: operating at the line rate, it is able to distribute the processing among stateless processors, which can dynamically join and leave the infrastructure at any time. Experimental tests show relevant outcomes intended as the ability to systematically saturate the I/O (network and disk), preserving reliable computations (at-least-once delivery policy).**

Keywords–distributed computing; high performance computing; data systems.

I. Introduction

This paper is an extended version of the paper "Chimera, A Distributed High-throughput Low-latency Data Processing and Streaming System" presented at SOFTENG 2017, The Third International Conference of Advances and Trends in Software Engineering, held in Venice, Italy, during April 2017 [1].

With the gigantic growth of information-sensing devices (Internet of Things) [2] such as mobile phones and smart devices, the predicted quantity of data produced far exceeds the capability of traditional information management techniques. To accommodate the left-shift in the scale [3], [4], new paradigms and architectures must be considered. The big data branch of computer science defines these big volumes of data and is concerned in applying new techniques to bring insights to the data and turn it into valuable business assets.

Modern data ingestion platforms distribute their computations horizontally [5] to scale the overall processing capability. The problem with this approach is in the way the distribution is accomplished: through distributed processors, prior to vertically move the data in the pipeline (i.e., between stages), they need coordination, generating horizontal traffic. This coordination is primarily used to accomplish reliability and delivery guarantees. Considering this, and taking into account the expected growth in compound volumes of data, it is clear that the horizontal exchanges represent a source of high pressure both for the network and infrastructure: the volumes of data supposed to flow vertically are amplified by a given factor due to the coordination supporting the computations, prior to any movement. Distributing computations and reducing the number of horizontal exchanges are complex challenges. If one was to state the problem, it would sound like: *to reduce the multiplicative factor in volumes of data to fulfill coherent computations, a new paradigm is necessary and such paradigm should be able to i. provide lightweight and stateful distributed processing, ii. preserve reliable delivery and, at the same time, iii. reduce the overall computation overhead, which is inherently introduced by the distributed nature*.

An instance of the said problem can be identified in predictive analytics [6], [7] for monitoring purposes. Monitoring is all about: *i.* actively producing synthetic data, *ii.* passively observing and correlating, and *iii.* reactively or pro actively spotting anomalies with high accuracy. Clearly, achieving correctness in anomaly detection needs the data to be ingested at line rate, processed on-the-fly and streamlined to polyglot storages [8], [9], with the minimum possible delay.

From an architectural perspective, an infrastructure enabling analytics must have a pipelined upstream tier able to *i.* ingest data from various sources, *ii.* apply correlation, aggregation and enrichment kinds of processing on the data, and eventually *iii.* streamline such data to databases. The attentive reader would argue about the ETL-like nature of such a platform, where similarities in the conception and organization are undeniable; ETL-like kind of processing is what is needed to reliably streamline data from sources to sinks. The way this is accomplished has to be revolutionary given the context and technical challenges to alleviate the consequences of exploding costs and maintenance complexity.

All discussed so far settled a working context for our team to come up with a novel approach to distribute the workload on processors, while preserving determinism and reducing the coordination traffic to a minimum. Chimera (the name Chimera has been used in [10]; the work presented in this paper addresses different aspects of the data ingestion) was born as an ultra-high-throughput processing and streamlining system able to ingest and process time series data [11] at line rate, preserving a delivery guarantee of at least once with an out of the box configuration, and exactly once with a specific and tuned setup. The key design tenets for Chimera were: *i.* low-latency operations, *ii.* deterministic workload sharding, *iii.* backpropagated snapshotting acknowledgements, and *iv.*

traffic persistence with on-demand replay. Experimental tests proved the effectiveness of those tenets, showing promising performance in the order of millions of samples processed per second with an easy to deploy and maintain infrastructure.

The remainder of this paper is organized as follows. Section II focuses on the state-of-the-art and related works, with an emphasis on the current technologies and solutions meanwhile arguing why those are not enough to satisfy the forecasts relative to the advent of the IoT and the incremental adoption of IPv6. Section III presents Chimera and its architectural internals, tier by tier, with a strong focus on design, enabling algorithms and the most relevant communication protocols. Section IV presents the simple fault model Chimera achieves. Section V presents the results from the experimental campaign conducted to validate Chimera and its design tenets. Section VI concludes this work and opens to future developments on the same track, while sharing a few lessons learned from the field.

II. Related Work

When it comes to assessing the state of the art of streamline platforms, a twofold classification can be approached: 1. *ETL* platforms originally designed to solve the problem of ingestion (used in the industry for many years, to support data loading into data warehouses [12]), and 2. *Analytics* platforms designed to distribute the computations serving complex queries on big data, then readapted to perform the typical tasks of ingestion too. On top of those, there are *hybrid platforms* that try to bring into play features from both categories.

The ETL paradigm [13] has been around for decades and is simple: data from multiple sources, distributed or not, is transformed into an internal format, usually more convenient to work with, then processed with the intent to correlate, aggregate and enrich with other sources; the data is eventually moved into one or multiple storages. Apart of commercial solutions, plenty of open-source frameworks have been widely adopted in the industry; it is the case of Mozilla Heka [14], Apache Flume and Apache Nifi [15], [16], [17]. Heka has been used as a primary ETL for a considerable amount of time, prior to being dismissed for its inherent design pitfalls: the single process, multi-threaded design based on green threads (Goroutines [18] are runtime threads multiplexed to a small number of system threads) had scalability bottlenecks that were impossible to fix without a complete redesign. In terms of capabilities, Heka provided valid supports: a set of customizable processors for correlation, augmentation and enrichment. Apache Flume and Apache Nifi are very similar in terms of conception, but different in the implementation: Nifi was designed with security and auditing in mind, along with enhanced control capabilities. Both Flume and Nifi can be distributed; they implement a multi-staged architecture common to Heka too. The design principles adopted by both solutions are based on data serialization and stateful processors. This require a large amount of computational resources as well as network round trips. The poor overall throughput makes them unsuited solutions for the stated problem.

On the other hand, analytics platforms adapted to ETL-like tasks are Apache Storm, Apache Spark and Apache Flink [19], [20]; all of them have a common design tenet: a task and a resource scheduler distribute computations on custom processors. The frameworks provide smart scheduling policies that invoke, at runtime, the processing logic wrapped into the custom processors. Such a design brings a few drawbacks: the most important resides in the need of heavyweight acknowledgement mechanisms or complex distributed snapshotting to ensure reliable and stateful computations. This is achieved at the cost of performance and throughput [21]. From [22], a significant measure of the message rate can be extrapolated from the first benchmark. Storm (best in class) is able to process approximately 250K messages/s with a level of parallelism of eight, meaning 31K messages/s per node with a 22% message loss in case of failure.

The hybrid category consists of platforms that try to bring the best of the two previous categories into sophisticated stacks of technologies; exemplar of this category is Kafka Streams [23], [24], a library for stream processing built on top of Kafka [25], which is unfortunately complex to setup and maintain. In distributed, Kafka heavily relies on ZooKeeper [26] to maintain the topology of brokers. Topic offset management and parallel consumers balancing depends on ZooKeeper too; clearly, a Kafka cluster needs at least a ZooKeeper cluster. However, Kafka Stream provides on average interesting levels of throughput.

As shown, three categories of platforms exist, and several authoritative implementations are provided to the community by as many open-source projects. Unfortunately, none of them is suitable to the given context and inherent needs.

III. Anatomy of Chimera

Clearly, a novel approach able to tackle and solve the weaknesses highlighted by each of the categories described in Section II is needed. Chimera is an attempt to remedy those weaknesses by providing a shared nothing processing architecture, moving the data vertically with the minimum amount of horizontal coordination and targeting *at-least-once delivery guarantee*. Chimera also offers a minimum of fault tolerance by being able to replay data in case of failures. However, as of time of writing, it is not resilient to byzantine failures.

Chimera is purely written in Java. The main reason behind this choice is because the project started as a proof of concept in the context of the first author master thesis. As such, given tight time constraints, Java has been picked as primary programming language for its higher productivity over higher performance languages. Productivity includes in particular facets such as the large amount of open-source libraries as opposed to the very proprietary aspect inherent to languages such as C and C++. The remainder of this section presents Chimera and its anatomy, intended as the specification of its key internals.

A. High Level Overview

Figure 1 presents Chimera by its component tiers. Chimera isolates three layers: *i.* queuing, *ii.* processing and *iii.* persistence. A coordinator, or cluster manager, also needs to be present for coordination purposes. To have Chimera working, it would therefore require at least three nodes, each of which assigned to one of the three layers, in addition to the cluster manager, which can by itself be a whole cluster if needed (one node is the bare minimum). Each node is focused on a specific task and only excels at that task. Multiple reasons drive such a decision. First, the separation of concerns simplifies the overall

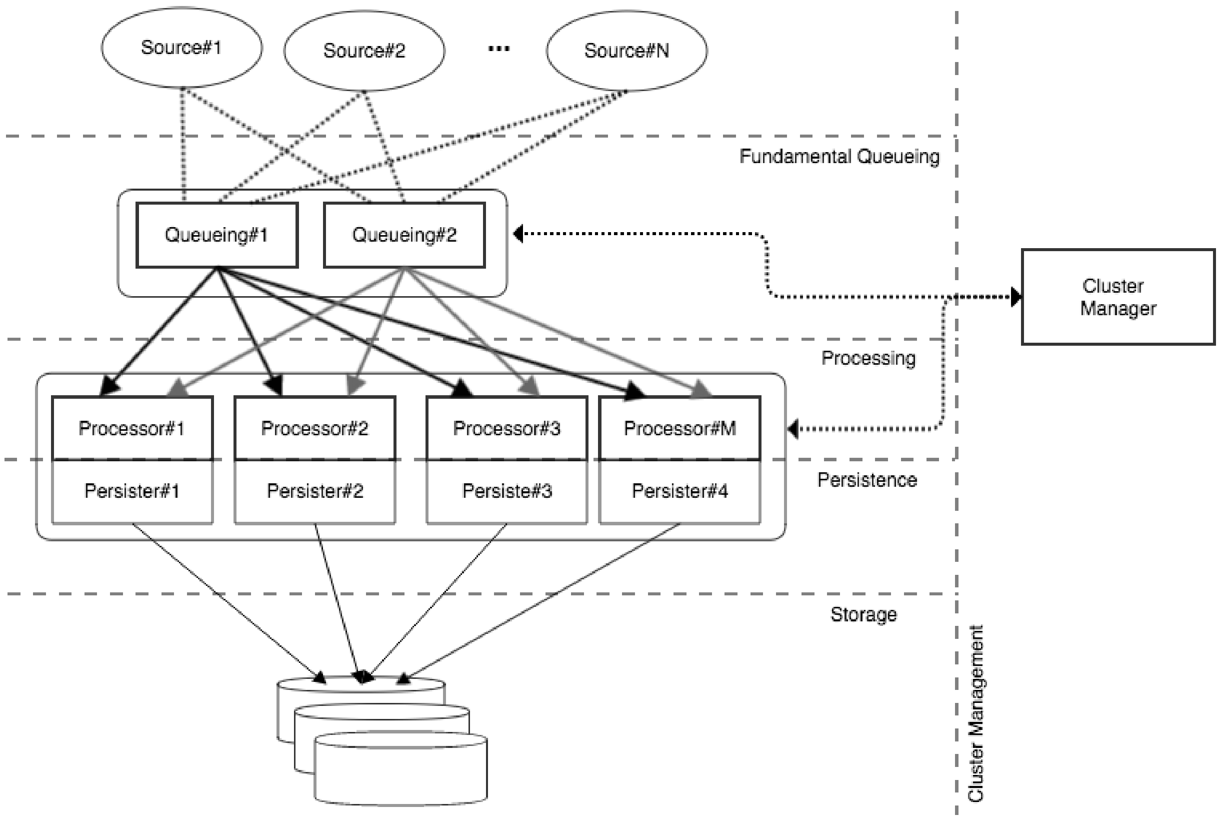


Figure 1. Chimera 10K feet view. Architectural sketch capturing the main tiers, their interactions, as well as relationships.

system, especially from an operational and maintainability perspective. Indeed, given the shared nothing architecture, joining an existing Chimera cluster only requires a handshake with the cluster manager. Secondly, in order to easily scale horizontally, organizing Chimera in a micro-service-like fashion was highly preferable. By distributing the tasks into independent nodes, scaling out only requires the addition of new nodes into the cluster, with performance expected to scale linearly with the number of nodes. Finally, this design enforces reliability by avoiding a single point of failure. Indeed, Chimera strives at providing a bare minimum in terms of fault tolerance. The whole system goes down only if there is no longer a node assigned to a particular tier, or if the coordinator goes down.

B. *Internal Data Model*

Data ingested by Chimera is first transformed into an internal data model. Designed for time series, the model used follows a scheme based on tags and fields, where the latter are actual metrics that provide the telemetric view for monitoring and alerting. Tags, on the other hand, can be seen as metadata. This schema is convenient for persistency, as it is a popular model used by time series databases, allowing such data to be smoothly persisted to such storages. In Chimera, tags and fields are simple key-value pairs, internally stored as ordered maps.

An important aspect of this data model is how a key is built from a data point (note that the term key here does not refer to the key from a key-value pair perspective). The key is an important component used to compute a hash, which is subsequently used for sharding. The key is obtained by concatenation of the tags, justifying the usage of ordered maps: two data points containing the same tags and values must give the same key.

C. *Fundamental Queuing Tier*

The fundamental queuing layer plays the central role of consistently sharding the inbound traffic to the processors (a processor refers to a node belonging to the processing tier). Consistency is achieved through a cycle-based algorithm, allowing dynamic join(s) and leave(s) of both queue nodes (a queue node refers to a node belonging to the fundamental queuing tier) and processors. To maintain statelessness of each component, the cluster manager ensures coordination between queue nodes and processors. Figure 2 gives a high-level view of a queue node.

Let $X = \{X_1, X_2, \dots, X_N\}$ be the inbound traffic, where N is the current total number of queue nodes. X_n denotes the traffic queue node n is responsible to shard. Let $Y = \{y_{11}, y_{12}, \dots, y_{1M}, y_{21}, \dots y_{2M}, \dots, y_{NM}\}$ where M is the current total number of processors. It follows that $X_n = \{y_{n1}, y_{n2}, \dots, y_{nM}\}$, y_{nm}, $n \in [1, N]$ and $m \in [1, M]$, is the traffic directed at processor m from queue node n. Note that Y_m is all the traffic directed at processor m, i.e., $Y_m = \{y_{1m}, y_{2m}, \dots, y_{Nm}\}$.

As suggested above, the sharding operates at two levels. The first one happens at the queue nodes. Each node n only accounts for a subset X_n of the inbound data, reducing the traffic over the network by avoiding sending duplicate data (note that each queue node still receives the totality of the traffic). X_n is determined by using a hash function on the key of the data d, i.e., $d \in X_n \iff hash(key(d)) \mod N = n$, where the function *key()* gives the key of a data point, as described in Section III-B. The second level of sharding operates at the processor level, where $\forall d \in X_n$, $d \in Y_m \iff hash(key(d)) \mod M$ = m. See Algorithm 1 for a synthetic view.

N and M are variables maintained by the coordinator, and each queue node keeps a local copy of these variables (to adapt the range of the hash functions). The coordinator updates N and M whenever a queue node joins/leaves, respectively a processor joins/leaves. This event also triggers a watch, which causes the coordinator to send a notification to all the queue nodes with the new value of N or/and M. However, the local values in each queue node are not immediately updated, rather it waits for the end of the current cycle. More details will follow in Section III-F.

A cycle can be defined as a batch of messages. This means that each y_{nm} belongs to a cycle c. Let us denote y_{nm_c} the traffic directed to processor m by queue node n during cycle c. Under normal circumstances (no failure), all the traffic directed at processor m (i.e., Y_m) will be received. Queue node n will advertise the coordinator that it has completed cycle c. Upon receiving all the data and successfully flushing it to the persistent storage, processor m will also advertise that cycle c has been properly processed and stored (see Algorithm 2 for a synthetic view of the tasks accomplished by a processor). As soon as all the processors advertised that they have successfully processed cycle c, the queue nodes move to cycle $c+1$ and start over.

Let us now consider a scenario with a failure. First, the failure is detected by the coordinator, which keeps track of live nodes by the mean of heartbeats. Let us assume the case of a processor m failing. By detecting it, the cluster manager adapts $M = M - 1$, and advertises this new value to all the queue nodes. The latter do not react immediately, but wait for the end of the current cycle c. At cycle $c+1$, the data that has been lost ($\forall d \in Y_m$) is resharded and sent over again to the new set of live processors. This is possible because all the data has been persisted by the journaler (see Section III-C3). This generalizes easily to more processors failing. See Algorithm 3.

Secondly, let us consider the case where a queue node fails during cycle c. A similar process occurs: the cluster manager notices that a queue node is not responsive anymore, and therefore adapts $N = N - 1$, before advertising this new value to the remaining queue nodes. At cycle $c = c + 1$, $\forall d \in X_n$ are resharded among the set of live queue nodes, and the data sent over again. Similarly, this generalizes to multiple queue nodes failing.

The case of queue node(s) / processor(s) joining is trivial. The node announces itself to the coordinator (handshake), which adapts the value of N or M to $N = N+1$ or $M = M+1$. This value is then advertised to all the queue nodes, which wait for the next cycle before taking it into consideration. The new queue node or processor is therefore idle until the start of a new cycle.

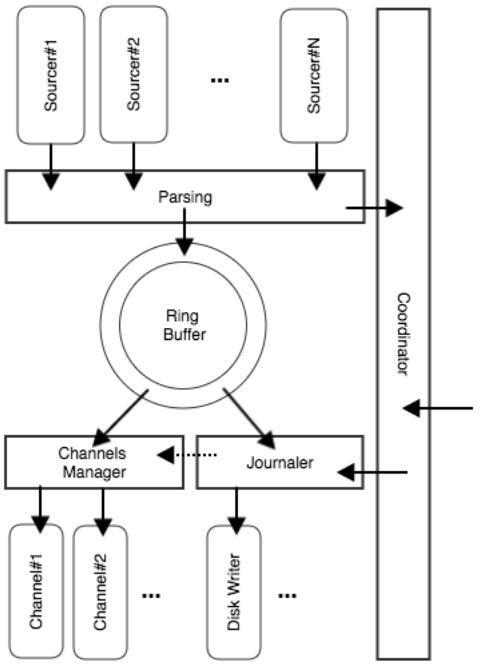


Figure 2. Fundamental queuing internals. A ring buffer disciplines the low-latency interactions between producers and consumers, respectively the sourcers that pull data from the sources, and the channels and journalers that perform the I/O for fundamental persistence and forwarding for processing.

Algorithm 1: Cyclic ingestion and continuous forwarding in the fundamental queuing tier.

```
Data: queueNodeId, N, M
Result: continuous ingestion and sharding.
initialization;
while alive do
    data = ringBuffer.next();
    n = hash(key(data)) mod N;
    if n == queueNodeId then
        m = hash(key(data)) mod M;
        push(data, m); // push in queue m
    end
    if endOfCycle then
        c = c+1;
        advertise(queueNodeId, c);
    end
end
```

Note that the approach described above guarantees that the data is delivered at least once. It however does not ensure exactly-once delivery. Section III-F3 complements the above explanations. Finally, recall that byzantine failures are out of scope and will therefore not be treated. It is worth emphasizing that introducing resiliency to such failures would most likely require a stateful protocol, which is exactly what Chimera avoids. The remainder of this section gives more details about all the components residing within the queuing layer.

1) Parser: A parser is a simple plugin component, which acts as the entry point to the pipeline. It handles the logic of transforming the input traffic into a unified internal format that is more convenient to work with. Recall from Section III-B.

2) Ring Buffer: The ring buffer is based on a multi-producers and multi-consumers scheme. As such, its design resolves around coping with high concurrency. It is an implementation of a lock-free circular buffer [27], which is able to guarantee sub-microsecond operations and, on average, ultra-high-throughput. In particular, the buffer avoids contended writes on the head of the buffer by separating producers and consumers by means of sequence numbers, where ordering is ensured by using memory barriers. Locking is avoided by using atomic operations. Furthermore, the ring buffer is backed by a constant sized pre-allocated array. The consequences of the pre-allocation is that the buffer becomes completely garbage free and mechanical sympathique. Indeed, allocating everything at once highly increases the probability of havings all the objects allocated within the same memory area, thus allowing cache striding, which highly increases cache hits. As discussed in [27], being cache friendly gives a substantial increase in performance.

3) Journaler: The journaler is a component dealing with disk I/O for durable persistence and is the first consumer from the ring buffer. Its task consists in persisting to disk the entire traffic, by consuming the data from the ring buffer. In general, I/O is a known bottleneck in high performance applications. To mitigate performance hit, the journaler, written in Java, uses memory-mapped file (MMFs) [28] and keeps garbage collection to a minimum by working off-heap.

A memory-mapped file is a segment of virtual memory that is mapped to a resouce (most of the time a file) presents on disk. In contrary to typical I/O where reading/writing from/to a file requires a system call, reading/writing from/to a memory-mapped file operates directly in main memory, by-passing the costly operations incurred by system calls (specifically context switches). Because memory-mapped files are accessed through the operating systems memory manager, the files are automatically partitioned into pages and accessed as needed: memory management is left to the operating system. From a system call perspective, creating a mapping is achieved via mmap(). The mmap() system call tells the kernel to map a parameterized amount of bytes of an object represented by a file descriptor into memory. Once a mapping is completed, the application can write into the file by the mean of the memory, rather than going through the write() system call. Using a memory-mapped file therefore avoids the extraneous copies that occurs when using the read() or write() system call, where data must be moved to and from a user-space buffer. These operations now directly operate in memory, which, from the application perspective, simply implies moving pointers, possibly avoiding fseek() calls at the same time.

On the other hand, off-heap programming particularly applies to programming languages where memory management is not handled by the programmer. Java is the most typical example, with its garbage collector. Programming off-heap simply means that objects are no longer allocated on the heap, but directly in the shared memory space. This concretely means that garbage collection will not clear these objects, leaving this task to the programmer himself (a simple analogy would be to code like in C++, but in Java: there is a need to manage memory manually). This greatly alleviates performance hits caused by garbage collection, especially during a full swipe. See [29] for detailed explanations on Java's garbage collector.

4) Channel Manager: Communications between queue nodes and processors are handled by the channel manager module, which is the second consumer from the ring buffer. This module handles the channels that are established between queue nodes and processors. It is moreover the component directly talking to the cluster manager, effectively handling the sharding of the inbound traffic. A channel is a custom implementation of a push-based client/server raw bytes asynchronous channel, able to work at line rate. It is a point to point application link and serves as an unidirectional pipe (from queue node to one instance of processor). Despite the efforts in designing the serialization and deserialization from scratch, the multi-threaded extraction module in the processor will prove to be the major bottleneck of the whole pipeline (refer to Section V).

In more details, the channel manager is a singleton intelligent unit that maintains a set of channels, each of which is connected to a processor. The unicity of the channel manager is paramount for simplicity and to keep coordination within the process to a minimum. Consumers constantly poll data from the ring buffer, and forward them to the channel manager. The latter proceeds to extract the hash from the data point, which is obtained by combining the tag from this data point (recall from Section III-B). The hash is then modded with N (number of queuing nodes) and the result compared to the current queue node ID. If the value obtained does not correspond to the one from the current queue node, the data point is simply dropped. Otherwise, the channel manager proceeds to compute the destination of this data point, consisting in modding the same hash to M, the current number of live processors. The data point is then properly forwarded to a queue, where a dispatcher will further asynchronously send it to the appropriate processor. This is extremely important for the sake of processing: data coming from a same source and that need to be aggregated together must always be sent over to the same processor. This is ensured by the determinism of the hash function. The queues are indexed and mapped directly to their respective processor ID, making it easy and fast to find the appropriate queue given the processor ID. Finally, the usage of queues here can further be justified as an application of the Half-Sync/Half-Async pattern [30], which promotes the integration of synchronous and asynchronous I/O for maximum efficiency in concurrent programming. The whole flow is synthetically summarized in Algorithm 1.

D. Shared-nothing Processing Tier

A processor is a shared-nothing process, able to perform a set of diversified lossless computations, in particular stateful aggregations. Recall that this is possible because a same processor is guaranteed to receive all the traffic matching a same key. Aggregation is an important step as one of nowadays big problems is that most databases are unable to cope with the traffic generated by modern distributed systems or infrastructures. The same applies for the querying aspect. Indeed, assuming there is somehow a way for a database to persist ten million data points per second, querying one hour worth of data would actually query $3.6 * 10^{10}$ data points, a query that would take a while time to return: unaffordable in

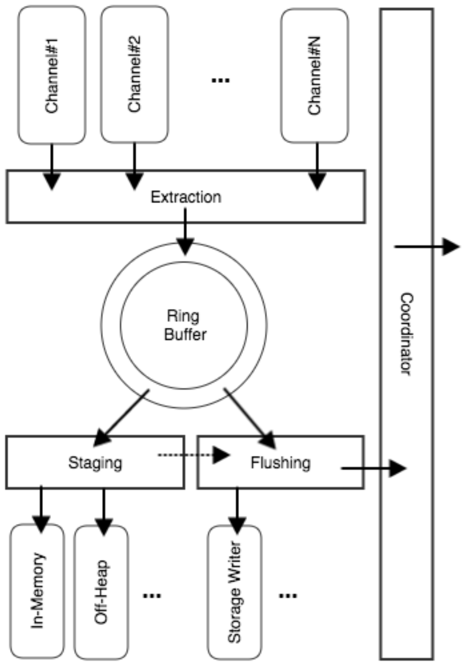


Figure 3. Processor internals. A ring buffer disciplines the interactions between producers and consumers, respectively the inbound channels receiving the data samples to process, and the staging and flushing sub-stages that store the data either for further processing or for durable persistence.

Algorithm 2: Cyclic reception, processing and flushing.

```
Data: processorId
Result: continuous processing and cyclic flushing.
initialization;
while alive do
    bytes = channel.receive();
    data = extract(bytes);
    processed = process(data);
    if to be staged then
        stage(processed);
    else
        flush();
        c = c+1;
        advertise(processorId, c);
    end
end
```

the context of monitoring where almost real-time responses are expected.

Internally, a processor is composed of three major components, which are the extractor, the stager and the flusher, as depicted on Figure 3. When joining an existing Chimera cluster, a processor only needs to advertise itself to the coordinator in order to start receiving traffic at the next useful cycle. Being stateless, it allows indefinite horizontal scaling. Details about the three main components of the processing tier are given below. The whole flow followed by a processor is synthetiacally summarized in Algorithm 2.

1) Extractor: The extractor module is a multi-threaded component that detaches asynchronous threads to rebuild the data received from the queue nodes into Chimera's internal model. It is the downstream of the channels (as per Section III-C4), which works with raw bytes for maximum efficiency.

In essence, the extractor is nothing but a big array of pre-allocated buffers. The size of the array is a static constant and is estimated by using Little's Law [31], rounded to the closest power of two for memory friendliness. The pre-allocation again favours byte continuity in memory and reduces garbage whereas the static size prevents unexpected memory exhaustion. Bytes received from the channel are sequentially allocated in the first empty buffer until completely filled, in which case the next buffer is used. If all the buffers are filled up, the extractor can effectively become a bottleneck. As soon as a buffer is completely filled up, a thread is detached and will asynchronously rebuild the data object(s) into Chimera's internal format. Upon completion of the reconstruction, the thread releases the buffer and returns it to the buffer pool, where it is made available for further bytes allocation.

2) Staging: The warehouse is the implementation of the staging area in Figure 3. It is an abstraction of an associative data structure in which the data is staged for the duration of a cycle; it is pluggable and has an on-heap and off-heap implementation. It supports various kinds of stateful processing, i.e., computations for which the output is function of a previously stored computation. In fact, as for most of the components in Chimera, the warehouse also has a pluggable side, providing flexibility, meaning any implementation fitting the needs can be used. As an example, the processor used for benchmarking Chimera has the inbound data aggregated on-the-fly for maximum efficiency; at the end of the cycle, all the data currently sitting in the warehouse get flushed to the database. However, partial data is not committed, meaning that unless all the data from a cycle c is received (i.e., Y_{m_c}), the warehouse will not flush.

In more details, the length of a cycle greatly impacts the warehouse. Indeed, the aggregation period is solely based on the length of a cycle, which can be defined by time or batch size. This is an important trade-off to make, as a longer cycle will likely means coarser data granularity, as more data will be aggregated before persisted to storage. This parameter requires fine tuning depending on the context. In the case of monitoring, a cycle will typically be fairly small, to keep maximum granularity and almost real-time data. Indeed, the longer the cycle, the later the data will be flushed to storage, which consequently means later query-able data.

3) Flusher: The flusher is an asynchronous component flushing the processed data into a storage of choice. It asynchronously consumes the data from the staging area, i.e. the warehouse, at the end of each cycle. The data is batched into a query that is submitted to the selected storage, and signals the end of a cycle to the cluster manager by acknowledging the data has been properly persisted. The flusher is a pluggable client that handle communications with the persistence layer, as queries differ depending on which database is used.

E. Persistence Tier

The persistence tier is a node of the ingestion pipeline that runs a database. This is the sole task of such kind of nodes. For the benchmarking, Chimera makes use of a time series database called KairosDB [32], which is built on top of Apache Cassandra [33]. At design time, the choice was made considering the expected throughput and the possibility to horizontally scale this tier too. Chimera offers a flexible plugin based model where selecting a different database is possible by simply providing a client talking the selected storage language (the flusher).

F. Core Protocols

The focus of this section is on the core protocols, synthetically and briefly formalized as the main algorithms implemented at the fundamental queuing and processor tiers. Their design targeted the distributed and shared nothing paradigm: coordination traffic is backpropagated and produced individually by every processor. The backpropagation of acknowledgements refers to the commit of the traffic shard emitted by the target processor upon completion of a flush operation (see Section III-D3). This commit is addressed to the coordinator only, avoiding horizontal coordination. To make sense of these protocols, the key concepts to be taken into consideration are *ingestion cycle* and *ingestion group*, as per their definitions.

1) Cyclic Operations: The ingestion pipeline works on ingestion cycles, which are configurable batching units; the overall functioning of the algorithm is independent of the cycle length, which may be an adaptive time window or any batching policy, ranging from a single unit to any number of units fitting the needs, context and type of data. Algorithm 1 presents the pseudo-code for the cyclic operations of Chimera on the fundamental queuing tier, which is mostly performed by the channel manager (Section III-C4), and Algorithm 2 presents the pseudo-code for the processing tier (Section III-D).

2) On-demand Replay: On-demand replay needs to be implemented in case of any disruptive events occurring in the ingestion group, e.g., a failed processor or queue node. In order to reinforce reliable processing, the shard of data originally processed by the faulty member needs to be replayed, and this has to happen on the next ingestion cycle. The design of the cyclic ingestion with replay mechanism allows to mitigate the effect of dynamic join and leave: the online readaptation only happens in the next cycle, without any impact on the current one. Recall from Section III-C3 that the traffic is persisted by the journaler.

Algorithm 3 presents the main flow of operations needed to make sure that any non committed shard of traffic is first re-processed consistently, and then properly flushed onto the storage. Note that this process of replaying can be nested in case of successive failures. It provides eventual consistency in the sense that the data will eventually be processed and persisted.

3) Dynamic Join/Leave: Any dynamic join(s) and leave(s) are automatically managed with the group membership and the distribution protocol. Join means any event related to a processor/queue node advertising itself to the cluster manager (or coordinator); instead, leave means any event related to a processor leaving the ingestion group and stop advertising itself to the cluster manager (e.g., a failure). Upon the arrival of a new processor, nothing happens immediately. Instead at the beginning of the next cycle, it is targeted with its shard of traffic; whenever a processor leaves the cluster, a missing commit for the cycle is detected and the on-demand replay is triggered to have the shard of traffic re-processed and eventually persisted by one of the live processors.

Algorithm 3: Data samples on-demand replay, upon failures (processor(s) not able to commit the cycle).

```
Data: cycleOfFailure, queueNodeId, prevM, M,
      failedProcessorId
Result: replay traffic according to the missing
        processor(s) commit(s).
initialization;
while alive do
    data = retrieve(cycleOfFailure);
    while data.hasNext() do
        curr = data.next();
        if hash(curr) mod N == queueNodeId then
            if (hash(curr) mod prevM) ==
              failedProcessorId then
                m = hash(curr) mod M;
                insert(curr, m);
            end
        end
    end
end
```

IV. Fault Model

This section presents a basic fault model for Chimera. The model stays shallow as the focus of this work is not security and fault resiliency, but rather scalability, performance and operational simplicity. Given a system deployed into a distributed environment, the system can be defined as a set of processes communicating between them by mean of the network. Thus, the two entities involved are the network and the processes. No distinction between the various processes is made, to stay as general as possible.

Considering three types of failures, which is reasonable given the nature of the system, the following presents whether Chimera tolerates them or not, and if yes, how. Getting inspiration from the taxonomy proposed in [34], Table I summarizes the failures (column) tolerated by Chimera, with respect to its components (row). Given the coordinator and the database are two full fledged external components, the focus will be on the queue nodes and the processors. Worth is to note that multiple failures could occur at the same time, clearly they are not mutually exclusive. Also, despite going as exhaustively as possible through all the possible failures, Chimera does not implement solutions to sustain them all. Only a small subset is addressed, as building a complete fault tolerant system was far beyond the scope of this work. Tolerance can be divided into three categories, each of which characterizes a particular dimension:

- Prevention: This dimension is concerned about ways of preventing a failure to occur. This encompasses, for example, good programming techniques and robust design.
- Detection: This deals with ways of detecting a failure occurred. It is particularly difficult in an asynchronous environment, where no assumption about time can be made.

- Mitigation: This is concerned about mitigating the damages and losses in case of failure. For example, right before failing, a process could attempt to persist its state on disk to ease the recovery.

Chimera will refer to these three dimensions to assess tolerance to given faults.

A. *Process - Crash*

From a process perspective, a crash could be considered the most drastic failures. Preventing a crash always start with clean coding and testing, reducing the chance of finding the process in an unexpected state. Crashes can also be induced by external events such as hardware failure, but this is irrevelant from a process perspective. Detecting that a process crashed in Chimera relies on the coordinator. Indeed, the coordinator will detect any unresponsive node(s) and act accordingly. Given the coordinator can be by itself a cluster, detecting that a coordinator node crashed is left to the coordinator itself, as ZooKeeper does it for example. In terms of mitigation, the queue nodes always persist the entirety of the traffic on disk via the journaler. In case of crash, the data is therefore recoverable with no loss. In case of the processors, they are totally stateless: there is no need to mitigate as nothing would be lost.

B. *Process - Corrupted Output*

A corrupted output failure can be divided into two causes: hardware or software. In the software case, preventing such a failure again starts with thorough testing to ensure the algorithms are behaving as expected. Test coverage and test generation are the difficulties here, some techniques are discussed in [35]. Input checking also plays a central role in prevention. A mean to detect a corrupted output is through data auditing. In the basic form, if an output is simply unexpected, exceptions handling should be put in place to avoid any crash.

In the case of corrupted output caused by hardware, such errors (bit flips for example) are simply assumed impossible to deal with from the application level. For example, it is discussed in [36] that CRCs are not enough to ensure error free packets.

C. *Network - Crash*

Network crash is completely unpredictable and very few people have control over the network (excluding malicious users). Replication and deploying Chimera in multiple independent regions would prevent a complete shut down of the system. Detecting a network crash is straightforward (no communication is possible within a certain region). Mitigating such an event consists in saving the state of each process until the network is available again. For Chimera, the queue nodes already do that, and as the processors are stateless, there is no state to save.

D. *Network - Corrupted Output*

A corrupted output caused by the network is in principle assumed to be prevented by TCP. This should therefore not affect the system at the application level.

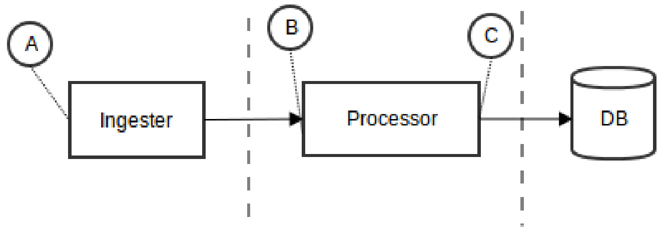


Figure 4. Graphical representation of the experimental methodology used to assess the performance of Chimera, tier by tier.

V. EXPERIMENTAL CAMPAIGN

In order to assess Chimera's performance with a focus to validate its design, a test campaign has been carried out. In this section, the performance figures are presented, notes are systematically added to give context to the figures and to share with the reader the observations from the implemented campaign.

A. *Testbench*

Performance testing has been conducted on a small cluster of three bare metal machines, each of which running with CentOS v7. Machines were equipped with two CPUs of six cores each, 48 GB of DDR3 and a HDD; they were connected by the mean of a 1 Gbit switched network. These are fairly small machines, which is perfectly aligned with one of the goals of Chimera, i.e., run on commodity hardware. Naturally, given this small cluster, one node was dedicated to the queuing tier, one node acted as a processor while the last one maintained the storage and played the role of cluster manager.

B. *Experiments*

The synthetic workloads were randomly generated, following a pre-defined schema reflecting the expected traffic from a production environment. The randomness might introduce slight unpredictabilities in the size of each data point, which are negligible given the load at which Chimera will operate, i.e. the unpredictabilies will be very small compared to the whole traffic. For each test scenario, a warm-up phase getting the JVM started is ran prior to running twenty iterations were results are being collected. A small idle period during each iteration is in place to clear any remaining garbage from a previous iteraiton. The results for each iteration were then averaged and summarized.

Figure 4 presents the testbench organization: probes were put in points A, B and C to capture relevant performance figures. As evident, the experiments were carried out with a strong focus on assessing the performance of each one of the composing tiers, in terms of inbound and outbound aggregated traffic.

The processor used for the tests performs a statistical aggregation of the received data points on per cycle basis; this was to alleviate the load on the database, which was not able to keep up with Chimera's average throughput. The statistical aggregation extracts typical measures such as max, min, mean and variance. The measures are extracted and aggregated on-the-fly as data points arrive [37], then flushed to storage at the

TABLE I. Summary of the faults tolerated by Chimera. The columns indicate the type of fault, the rows refer to the component involved in Chimera. Process indicate any process in Chimera, i.e. processor or queue node.

	Crash	*Corrupted Output*
Process	Yes	Yes
Network	Yes	Yes

end of a cycle, i.e. batch of datapoints. The remainder of this section presents the results with reference to this methodology.

C. Results

1) Fundamental Queuing Inbound Throughput:

a) Parsing: At the very entrance of any pipeline sits the parsing submodule, which is currently implemented following a basic scheme. This is mostly because the parsing logic highly relates to the kind of data that would be ingested by the system. As such, parsing optimizations can only be carried out when actual data is pumped into Chimera. Nevertheless, stress testing has been conducted to assess the performance of a general purpose parser. The test flow is as follow: synthetic workloads is created and loaded up in memory, before being pumped into the parsing module, which in turn pushes its output to the ring buffer. The results summarized in Table II are fairly good: a single threaded parsing submodule was able to parse 712K messages per second, on average. Clearly, as soon as the submodule makes use of multiple threads, the parser was able to saturate the ring buffer capacity.

b) Ring Buffer: The synthetic workload generator simulated many different sources pushing messages of 500 bytes (with a slight variance due to randomness) on a multi-threaded parsing module. In order to push the limits of the actual implementation, the traffic was entirely loaded in memory and offloaded to the ring buffer. The results were fair, the ring buffer was always able to go over 4M data samples ingested per second; a summary of the results as a function of the input bulks is provided in Figure 5(a);

c) Journaler: As specified in Section V, the testbench machines were equipped with HDDs, clearly the disk was a bottleneck, which systematically induced backpressure to the ring buffer. Preliminary tests using the HDD confirmed the hypothesis: the maximum I/O throughput possible was about 115 MByte/s. That was far too slow considering the performance Chimera strives to achieve. As no machine with a Solid State Drive (SSD) was available, the testing was carried out on the temporary file system (`tmpfs`, which is backed by the memory) to emulate the performance of an SSD. Running the same stress tests, a write throughput of around 1.6 GByte/s has been registered. By the time of writing, the latter is a number achieved by a good SSD [38], and which is perfectly in line with the ring buffer experienced throughput (approx. 2 GByte/s of brokered traffic data). Figure 5(b) gives a graphical representation of the results.

2) Fundamental Queuing Outbound Throughput:

a) Channel: Results from channel stress testing are shown in Figure 6(a). The testbench works on bare metal machines on a 1 Gbit switched network, which is, as for the case of the HDD, a considerable bottleneck for Chimera. Over the network, 220K data points per second were transferred (approx. 0.9 Gbit/s), maxing out the network bandwidth. Stress tests were repeated with a local setup, approaching the same reasoning as per the case of journaler. The results are reported in Figure 6(b), which demonstrates the ultra high-level of throughput achievable by the outbound submodule of the fundamental queuing tier: the channel keeps up with the ring buffer, being able to push up to 4M data points per second.

3) Processor Inbound Throughput:

a) Channel: The channel is a common component, which acts as sender on the queuing side, and as receiver on the processor side. The performance to expect has already been assessed, so for the inbound throughput of the processor the focus would be on the warehouse, which is a fundamental component for stateful processing. Note that processors operate in a stateless way, meaning that they can join and leave dynamically, but, of course, they can perform stateful processing by staging the data as needed and as by design of the custom processing logic.

b) Staging Area: Assessing the performance of this component was critical to shape the expected performance curve for a typical processor. The configuration under test made use of an on-heap warehouse (see Section III-D2), which guarantees a throughput of 3.5M operations per second, as shown on Figure 7(a). Figure 7(b) shows the result obtained from a similar test, but under concurrent writes; going off-heap was proven to be overkilling as further serialization and deserialization were needed, clearly slowing down the entire inbound stage of the processor to 440K operations per second. Note that with a decent serialization framework and an optimized writing strategy, the expected throughput of an off-heap data structure should easily outperform that of an on-heap approach.

c) Extractor: This module was proven to be the biggest bottleneck of Chimera. It has to deserialize the byte stream and unmarshal it into Chimera's domain object, using the default Java serializer. The multi-threaded implementation was able to go up to 0.9M data points rebuilt per second: a high backpressure was experienced on the channels pushing data at the line rate, producing high GC overhead on long runs. The adoption of a decent serialization framework would definitely improve the performance of the extractor by at least an order of magnitude.

4) Processor Outbound Throughput:

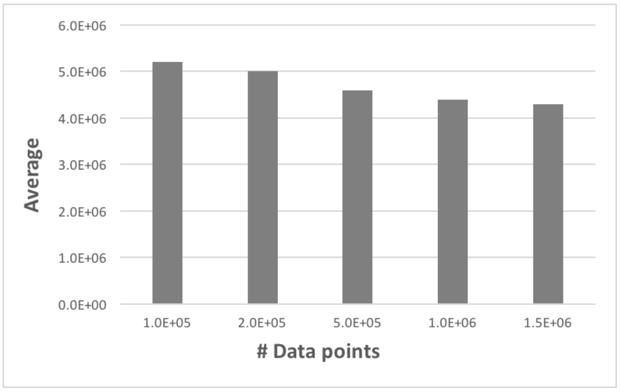
a) Flusher: It was very related to the specific aggregating processor and it was assessed to be approx. 85 MByte/s, which is reasonable considered the aggregation performed on the data falling into a batching on the cycle. The characteristic of this tier may variate with the support used for the storage.

D. Discussion

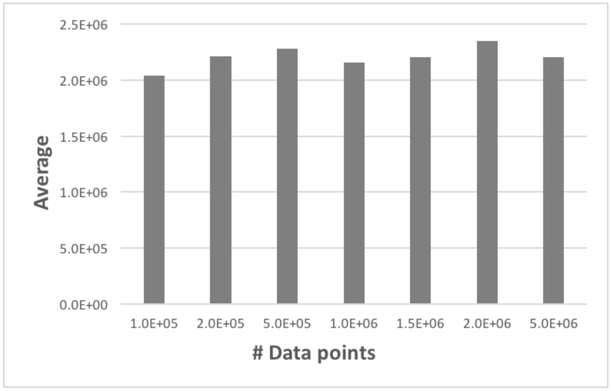
The test campaign was aimed at pushing the limits of each single module of the staged architecture. The setup put in

TABLE II. Summary of the experienced throughputs in millions per second. This table provides a quantitative characterization of Chimera as composed by its two main stages and inherent submodules. Parsing and extraction were multi-threaded, using a variable pool of cached workers (up to the limit of $(N * 2 + 1)$ where N was the number of CPUs available). Tests were repeated with a local processor to overcome the 1 Gbit network link saturation problem. The results involving the network are shown in the light gray shaded rows.

	Queuing [M/s]		*Processing [M/s]*			
Direction	Parsing	Ring Buffer	Journaler	Channel	Extraction	Staging
Inbound	6	4.3	4.3	0.2	0.2	0.2
Outbound	4.3	4.3	3.7	0.2	0.2	0.2
Inbound	6	4.3	4.3	4.3	0.9	0.9
Outbound	4.3	4.3	3.7	4.2	0.9	0.9



(a) Ring buffer stress test results. Synthetic traffic was generated as messages of average size 500 bytes.



(b) Journaler stress test results. Synthetic traffic was as per ring buffer.

Figure 5. Performance of the ring buffer and journaler.

place was a single process both for the fundamental queuing and processor tiers, so the performance figures showed in the previous sections were referring to such setup.

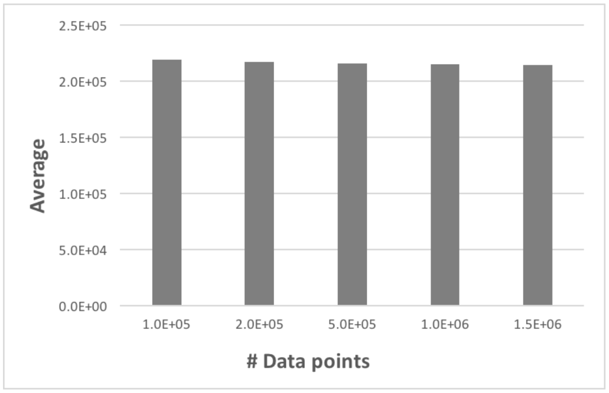
The experimental campaign has confirmed the ideas around the design of Chimera. As per Table II, Chimera is a platform able to handle millions of data samples flowing vertically in the pipeline, with a basic setup consisting of single queuing and processing tiers. No test have been performed with scaled setups (i.e., several queuing components and many processors), but considered the almost shared nothing architecture targeted for the processing tier (slowest stage in the pipe having the bottleneck in the extraction module), a linear scalability is expected, as well as a linear increase of the overall throughput as the number of processors grows up.

During the test campaign, resource thrashing phenomenon was observed [39]. The journaler pushed the write limits of the HDD, inducing the exhaustion of the kernel direct memory pages. The HDD was only able to write at a rate of 115 MByte/s, and therefore, during normal runs, the memory gets filled up within a few seconds, inducing the operating system into a continuous swapping loop, bringing in and out virtual memory pages.

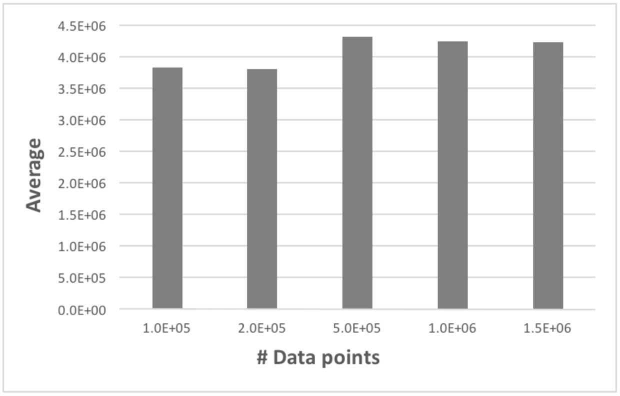
Figure 8 presents a plot of specific measurements to confirm the resource thrashing hypothesis. The tests consisted in writing over several ingestion cycles a given amount of Chimera data points to disk, namely one and three millions per cycle. The case of one million data points per batch shows resource thrashing after seven cycles: write times to HDD bump up considerably, the virtual memory stats confirmed pages being continuously swapped in and out; the case of three millions data points per batch shows resource thrashing after only two cycles, which is expected. High response times were caused by the cost of flushing the data currently in memory to the slow disk, meanwhile the virtual direct memory was filled up and swapped in and out by the kernel to create room for new data, as confirmed in [40].

VI. Conclusion

Chimera is a prototype solution implementing the proposed ingestion paradigm, which is able to distribute the queuing (intended as traffic persistence and replay) and processing tiers into a vertical pipeline, horizontally scaled, and sharing nothing among the processors (control flow is vertical, from queuing to processors, and from processors to queuing). The innovative distribution protocols allow to implement the backpropagated incremental acknowledgement, which is a key aspect for the delivery guarantee of the overall infrastructure: in case of failure, a targeted replay can redistribute the data on the live processors and any newly joining one(s). This same mechanism allows to redistribute the load, in case of backpressure, on newly joining members with a structured approach: the redistribution is implemented on a cyclic basis, meaning that a newly joined processor, once bootstrapped, starts receiving traffic only during the next useful ingestion cycle. This innovative approach solves the problems highlighted with the solutions currently adopted in the industry, keeping the level of complexity of the overall infrastructure very low: the decoupled nature of the queuing and processing tiers, as well as the backpropagation mechanism are as many design

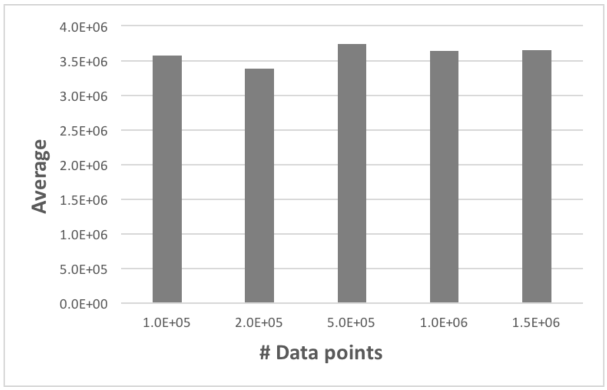


(a) Channel stress test results. Synthetic traffic was pulled from the ring buffer and pushed on the network, targeting the designated processor.

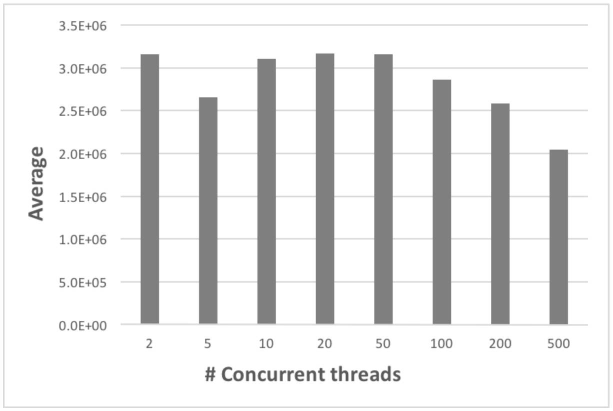


(b) Channel stress test results. Synthetic traffic was pulled from the ring buffer and pushed on the network, targeting the designated localhost processor.

Figure 6. Performance of the channel.



(a) Warehouse (i.e., staging area) stress test results. Scenario with non-concurrent writes.



(b) Warehouse (i.e., staging area) stress test results. Scenario with concurrent writes.

Figure 7. Performance of the staging area.

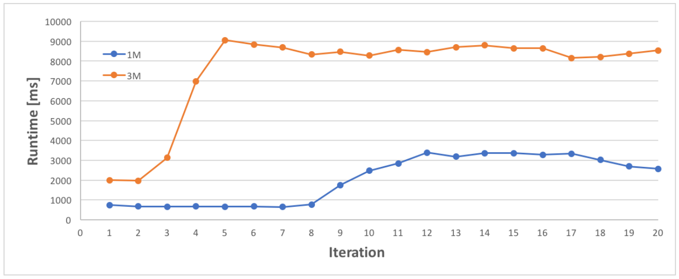


Figure 8. Experimented HDD-induced thrashing phenomenon. The I/O bottleneck put backpressure on the kernel, inducing high thrashing, which was impacting the overall functioning of the machine.

tenets that enable easy distribution and guarantee reliability despite the very high level of overall throughput.

From a performance standpoint, experimental evidences demonstrate that Chimera is able to work at line rate, maxing out the bandwidth. The queuing tier outperforms the processing tier: on average a far less number of CPU cycles is needed to first transform and second persist the inbound traffic, and this is clear if compared to the kind of processing described as example from the experimental campaign.

Finally, Chimera provides a few mechanisms to deal with failures. As explained, the queue nodes persist at maximum throughput the inboud traffic on disk for replay. The statelessness feature of the processors grant failure resiliency to Chimera out-of-the-box at this level, as the replay mechanism is all that is needed to overcome a failing processor. All in all, besides byzantine failures, Chimera yields good tolerance to failures simply by design.

A. Lessons Learned

The journey to design, implement and validate experimentally the platform was long and arduous. A few lessons have been learned by engineering for low-latency (to strive for the best from the single process on the single node) and distributing by sharing almost nothing (coordinate the computations on distributed nodes, by clearly separating the tasks and trusting deterministic load sharding). First lesson might be summarized as: *serialization is a key aspect in I/O (disk and network)*, a slow serialization framework can compromise the throughput of an entire infrastructure. Second lesson might summarized as:*memory allocation and deallocation are the evil in managed languages*, when operating at line rate, the backpressure from the automated garbage collector can jeopardize the performances, or worse, kill nodes (in the worst case, a process crash can be induced). Third lesson might be summarized as: *achieving shared nothing architecture is a chimera (i.e., something unique) by itself*, meaning that it looks almost impossible to let machines collaborate/cooperate without any sort of synchronization/snapshotting. Forth and last lesson might be summarized as: *tiering vertically allows to scale but it inevitably introduces some coupling*, this was experienced with the backpropagation and the replay mechanism in the attempt to ensure stateless and reliable processors.

B. Future Work

The first step into improving Chimera would be to work on a better serialization framework. Indeed, as shown in the test campaign, bottlenecks were found whenever data serialization comes into play. Existing open-source frameworks are available, such as Kryo [41] for Java. Secondly, in order to further assess the performance of Chimera, it would be necessary to run a testbench where multiple queue nodes and processors are live. Indeed, the test campaign has only been focused on one queue node, one processor, one storage node and a single node coordinator. This would also allow to further assess Chimera's resiliency to failures, and recovery mechanisms. Indeed, Byzantine failures have been excluded from the scope of this work, but resiliency with respect to such failures are necessary to enforce robustness and security. In particular, using Netflix Simian Army would be most interesting to assess the recovery mechanisms.

VII. Acknowledgement

This work has been carried out in collaboration with the École Polytechnique Fédérale de Lausanne (EPFL) as partial fulfillment of the first author master thesis work. A special thank goes to Prof. Katerina Argyraki for her valuable mentoring and her continuous feedback.

References

[1] L. Pascal and M. Paolo, "Chimera, a distributed high-throughput low-latency data processing and streaming system," IARIA, SOFTENG, 2017.

[2] D. Evans, "The Internet of Things," Cisco, Inc., Tech. Rep., 2011.

[3] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," Computer networks, vol. 54, no. 15, 2010, pp. 2787–2805.

[4] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (iot): A vision, architectural elements, and future directions," Future generation computer systems, vol. 29, no. 7, 2013, pp. 1645–1660.

[5] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," IEEE Transactions on Parallel and Distributed Systems, vol. 25, no. 6, 2014, pp. 1447–1463.

[6] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact." MIS quarterly, vol. 36, no. 4, 2012, pp. 1165–1188.

[7] P. Russom et al., "Big data analytics," TDWI best practices report, fourth quarter, 2011, pp. 1–35.

[8] M. Fowler and P. Sadalage, "Nosql database and polyglot persistence," Personal Website: http://martinfowler. com/articles/nosql-intro-original. pdf, 2012.

[9] A. Marcus, "The nosql ecosystem," The Architecture of Open Source Applications, 2011, pp. 185–205.

[10] K. Borders, J. Springer, and M. Burnside, "Chimera: A declarative language for streaming network traffic analysis." in USENIX Security Symposium, 2012, pp. 365–379.

[11] D. R. Brillinger, Time series: data analysis and theory. SIAM, 2001.

[12] R. Kimball and J. Caserta, The Data Warehouse? ETL Toolkit: Practical Techniques for Extracting, Cleaning, Conforming, and Delivering Data. John Wiley & Sons, 2011.

[13] P. Vassiliadis, "A survey of extract–transform–load technology," International Journal of Data Warehousing and Mining (IJDWM), vol. 5, no. 3, 2009, pp. 1–27.

[14] "Introducing Heka," https://blog.mozilla.org/services/2013/04/30/introducing-heka/, 2017, [Online; accessed 3-March-2017].

[15] D. Namiot, "On big data stream processing," International Journal of Open Information Technologies, vol. 3, no. 8, 2015, pp. 48–51.

[16] C. Wang, I. A. Rayan, and K. Schwan, "Faster, larger, easier: reining real-time big data processing in cloud," in Proceedings of the Posters and Demo Track. ACM, 2012, p. 4.

[17] J. N. Hughes, M. D. Zimmerman, C. N. Eichelberger, and A. D. Fox, "A survey of techniques and open-source tools for processing streams of spatio-temporal events," in Proceedings of the 7th ACM SIGSPATIAL International Workshop on GeoStreaming. ACM, 2016, p. 6.

[18] R. Pike, "The go programming language," Talk given at Googles Tech Talks, 2009.

[19] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas, "Apache flink: Stream and batch processing in a single engine," Data Engineering, 2015, p. 28.

[20] S. Kamburugamuve and G. Fox, "Survey of distributed stream processing," http://dsc.soic.indiana.edu/publications, 2016, [Online; accessed 3-March-2017].

[21] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng et al., "Benchmarking streaming computation engines: Storm, flink and spark streaming," in Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International. IEEE, 2016, pp. 1789–1792.

[22] M. A. Lopez, A. Lobato, and O. Duarte, "A performance comparison of open-source stream processing platforms," in IEEE Global Communications Conference (Globecom), Washington, USA, 2016.

[23] "Kafka Streams," http://docs.confluent.io/3.0.0/streams/, 2017, [Online; accessed 3-March-2017].

[24] "Introducing Kafka Streams: Stream Processing Made Simple," http://bit.ly/2nASDDw, 2017, [Online; accessed 3-March-2017].

[25] J. Kreps, N. Narkhede, J. Rao et al., "Kafka: A distributed messaging system for log processing," in Proceedings of the NetDB, 2011, pp. 1–7.

[26] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems." in USENIX annual technical conference, vol. 8, 2010, p. 9.

[27] M. Thompson, "Lmax disruptor. high performance inter-thread messaging library."

[28] S. T. Rao, E. Prasad, N. Venkateswarlu, and B. Reddy, "Significant performance evaluation of memory mapped files with clustering algorithms," in IADIS International conference on applied computing, Portugal, 2008, pp. 455–460.

[29] S. Microystems, "Memory management in the java hotspot virtual machine," 2006.

[30] D. C. Schmidt and C. D. Cranor, "Half-sync/half-async," Second Pattern Languages of Programs, Monticello, Illinois, 1995.

[31] J. D. Little, "Little's law as viewed on its 50th anniversary," Operations Research, vol. 59, no. 3, 2011, pp. 536–549.

[32] "KairosDB," https://kairosdb.github.io/, 2015, [Online; accessed 3-March-2017].

[33] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," ACM SIGOPS Operating Systems Review, vol. 44, no. 2, 2010, pp. 35–40.

[34] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," IEEE transactions on dependable and secure computing, vol. 1, no. 1, 2004, pp. 11–33.

[35] I. G. Harris, “Fault models and test generation for hardware-software covalidation,” IEEE Design & Test of Computers, vol. 20, no. 4, 2003, pp. 40–47.

[36] Y. Zhou, V. Lakamraju, I. Koren, and C. M. Krishna, “Software-based failure detection and recovery in programmable network interfaces,” IEEE Transactions on Parallel and Distributed Systems, vol. 18, no. 11, 2007.

[37] T. Finch, “Incremental calculation of weighted mean and variance,” University of Cambridge, vol. 4, 2009, pp. 11–5.

[38] “Intel SSD Data Center Family,” http://intel.ly/2nASMqy, 2017, [Online; accessed 3-March-2017].

[39] P. J. Denning, “Thrashing: Its causes and prevention,” in Proceedings of the December 9-11, 1968, fall joint computer conference, part I. ACM, 1968, pp. 915–922.

[40] L. Wirzenius and J. Oja, “The linux system administrators guide,” versión 0.6, vol. 2, 1993.

[41] “Kyro Serialization Framework,” https://github.com/EsotericSoftware/kryo, 2017, [Online; accessed 5-April-2017].