Chimera: A Distributed High-throughput Low-latency Data Processing and Streaming System

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 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 while processing in real-time. 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 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

With the gigantic growth of information-sensing devices (Internet of Things) [1] 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 [2], [3], 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 [4] 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 is 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 is a complex challenge. 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 [5], [6] for monitoring purposes. Monitoring is all about: *i.* actively produce synthetic data, *ii.* passively observe and correlate, and *iii.* reactively or pro actively spot 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 storage [7], [8], 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; however, 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 [9]; 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 [10] 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, with an emphasis on the

Copyright (c) IARIA, 2017.

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, with a strong focus on the enabling tiers and the most relevant communication protocols. Section IV presents the results from the experimental campaign conducted to validate Chimera and its design tenets. Section V 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 (used in the industry for many years, to support data loading into data warehouses [11]), 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 [12] has been around for decades and is simple: data from multiple sources is transformed into an internal format, then processed with the intent to correlate, aggregate and enrich with other sources; the data is eventually moved into a storage. Apart of commercial solutions, plenty of open-source frameworks have been widely adopted in the industry; it is the case of Mozilla Heka [13], Apache Flume and Nifi [14], [15], [16]. 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 [17] are runtime threads multiplexed to a small number of system threads) had scalability bottlenecks that were impossible to fix without a re-design. In terms of capabilities, Heka provided valid supports: a set of customizable processors for correlation, augmentation and enrichment. Apache Flume and Nifi are very similar in terms of conception, but different in terms of implementation: Nifi was designed with security and auditing in mind, as well as 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, Spark and Flink [18], [19]; all of them have a common design tenet: a task and resource scheduler distributes 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 [20]. From [21], a significant measure of the message rate can be extrapolated from the first benchmark. Storm (best in class) is able to process approx. 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.

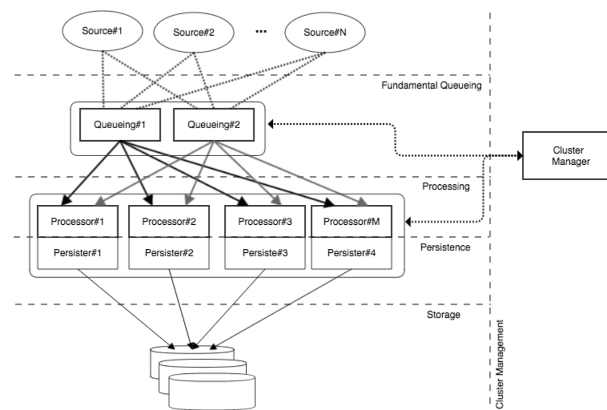


Figure 1. Chimera 10K feet view. Architectural sketch capturing the main tiers, their interactions, as well as relationships.

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 [22], [23], a library for stream processing built on top of Kafka [24], which is unfortunately complex to setup and maintain. In distributed, Kafka heavily uses ZooKeeper [25] to maintain the topology of brokers. Topic offset management and parallel consumers balancing relies 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*. 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. To have Chimera working, we would therefore need at least three nodes, each of which assigned to one of the three layers. 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 system. Second, it was a requirement to have something easy to scale out (by distributing the tasks into independent nodes, scaling out only requires the addition of new nodes). Finally, reliability: avoiding a single point of failure was a key design aspect.

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-553-1

B. Fundamental Queuing Tier

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

Let $X = \{X_1, X_2, \ldots, X_N\}$ 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 $Y = \{y_{11}, y_{12}, \ldots, y_{1M}, y_{21}, \ldots y_{2M}, \ldots, y_{NM}\}$, with M the current total number of processors, be the data moving from queue node to processors. It follows that $X_n = \{y_{n1}, y_{n2}, \ldots, y_{nM}\}$, where 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., $Y_m = \{y_{1m}, y_{2m}, \ldots, y_{Nm}\}$.

As suggested above, the sharding operates at two levels. The first one operates 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. X_n is determined using a hash function on the data d (data here means a sample, a message, or any unit of information that needs to be processed), i.e., $d \in X_n \iff hash(d) \mod N = n$. The second sharding operates at the processor level, where $\forall d \in X_n$, $d \in Y_m \iff hash(d) \mod M$ = m. See Algorithm 1.

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 triggers a watch: the coordinator sends a notification to all the queue nodes, with the updated values for N and/or M. However, the local values in each queue node is not immediately updated, rather it waits for the end of the current cycle. A cycle can be defined as a batch of messages. This means that each y_{nm} belongs to a cycle. 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 during cycle c (i.e., Y_{m_c}) will be received. Queue node n will advertise the coordinator that it has completed cycle c (see Algorithm 2). Upon receiving all the data and successfully flushing it, processor m will also advertise that cycle c has been properly processed and stored. As soon as all the processors advertise 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 coordinator 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 during cycle c ($\forall d \in Y_{m_c}$) are 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. This generalizes easily to more processors failing. See Algorithm 3.

Secondly, let us consider the case where a queue node

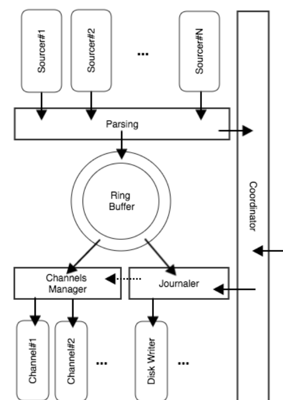


Figure 2. Fundamental queuing internals. A ring buffer disciplines the low-latency interactions between producers and consumers, respectively the sourcers that pull data from the sources, and the channels and journalers that perform the I/O for fundamental persistence and forwarding for processing.

fails during cycle c. A similar process occurs: the coordinator 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 $c = c + 1$, $\forall d \in X_{n_c}$ 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 and will therefore not be discussed here.

Note that the approach described above ensures that the data is guaranteed to be delivered at least once. It however does not ensure exactly-once delivery. Section III-E3 complements the above explanations.

Byzantine failures [26] are out of scope and will therefore not be treated. It is worth emphasizing that introducing resiliency to such failures would require a stateful protocol, which is exactly what Chimera avoids. Below, details about the three main components of the fundamental queuing tier are given.

1) 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.

2) Journaler: The journaler is a component dealing with disk I/O for durable persistence. In general, I/O is a known bottleneck in high performance applications. To mitigate performance hit, the journaler uses memory-mapped file (MMFs) [28].

3) Channel: Communications between fundamental queuing and processors is implemented via the channel module, which 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 (queuing tier to one instance of processor). Despite the efforts in designing the serialization and deserialization from scratch, the extraction module in the processor will prove to be the major bottleneck (refer to Section IV).

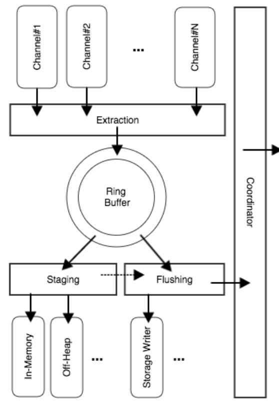


Figure 3. Processor internals. A ring buffer disciplines the interactions between producers and consumers, respectively the inbound channels receiving the data samples to process, and the staging and flushing sub-stages that store the data either for further processing or for durable persistence.

C. Shared-nothing Processing Tier

A processor is a shared-nothing process, able to perform a set of diversified computations. It is composed of three major components, which are the extractor, the stager and the flusher, as depicted on Figure 3. A processor only needs to advertise itself to the coordinator in order to start receiving traffic at the next cycle. Being stateless, it allows indefinite horizontal scaling. Details about the two main components of the processing tier are given below.

1) Extractor: The extractor module is the component that asynchronously rebuilds the data received from the queue nodes into Chimera's internal model. It is the downstream of the channel (as per Section III-B3).

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. 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 gets 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.

D. 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. Chimera makes use of a time series database (TSDB) [29] built on top of Apache Cassandra. At design time, the choice was made considering the expected throughput and the possibility to horizontally scale this tier too.

E. Core Protocols

The focus of this section is on the core protocols, intended as the main algorithms implemented at the fundamental queuing and processor components; their design targeted the

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

```
Data: queueNodeId, N, M
Result: continuous ingestion and sharding.
while alive do
    curr = buffer.next();
    n = hash(current) mod N;
    if n == queueNodeId then
        m = hash(curr) mod M;
        send(curr, m);
    end
    if endOfCycle then
        c = c+1;
        advertise(queueNodeId, c);
    end
end
```

Algorithm 2: Cyclic reception, processing and flushing.

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

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 operations. This commit is addressed to the coordinator only. 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 is independent of the cycle length, which may be an adaptive time window or any batch 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, and Algorithm 2 presents the pseudo-code for the processing tier.

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

Copyright (c) IARIA, 2017.
ISBN: 978-1-61208-553-1

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
        current = data.next();
        if hash(current) mod N == queueNodeId then
            if (hash(current) mod prevM) ==
             failedProcessorId then
                m = hash(current) mod M;
                send(curr, m);
            end
        end
    end
end
```

happens in the next cycle, without any impact on the actual one.

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.

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. 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 (e.g., a failure), 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.

IV. Experimental Campaign

In order to assess Chimera 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 runs on 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.

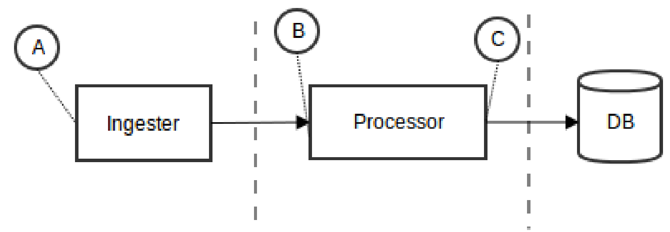


Figure 4. Graphical representation of the experimental methodology used to assess the performance of Chimera, tier by tier.

B. Experiments

The synthetic workloads were generated randomly. The data was formatted to reflect the expected traffic in a production environment. For each test scenario, twenty iterations were run; 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 remainder of this section presents the results with reference to this methodology.

C. Results

1) Fundamental Queuing Inbound Throughput:

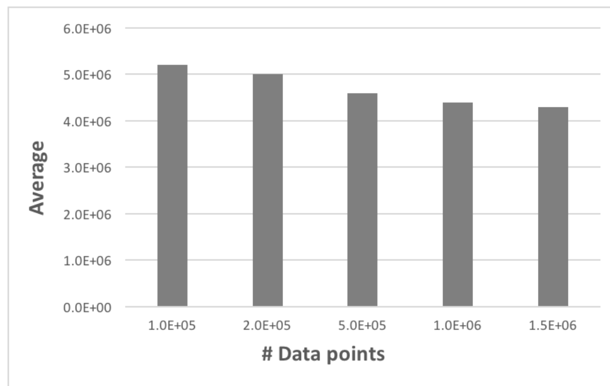
a) Parsing: At the very entrance of any pipeline sits the parsing submodule, which is currently implemented following a basic scheme. This is mostly because the parsing logic highly relates to the kind of data that would be ingested by the system. As such, parsing optimization can only be carried out 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 turns pushes its output to the ring buffer. The results summarized in Table I 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 byte (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 5(a);

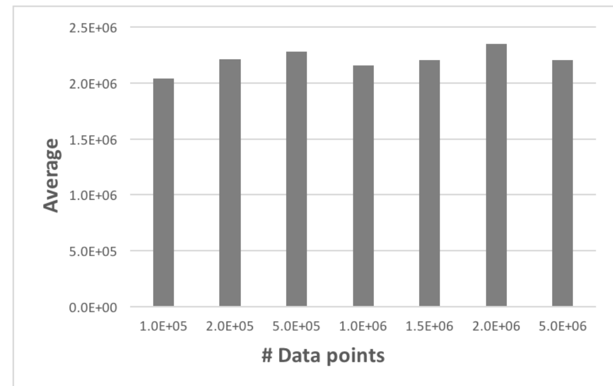
Copyright (c) IARIA, 2017.

TABLE I. 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 Byte.



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

Figure 5. Performance of the ring buffer and journaler.

c) Journaler: As specified in Section IV, 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 were 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 [30], and which is perfectly in line with 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 demonstrate 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:

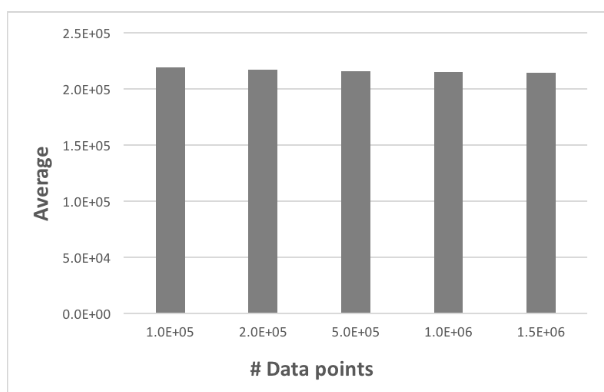
a) Channel: The channel is a common component, which acts as sender on the queuing side, and as receiver on the processor side. The performance to expect has already been assessed, so for the inbound throughput of the processor the focus would be on the warehouse, which is a fundamental component for 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-C2), 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.

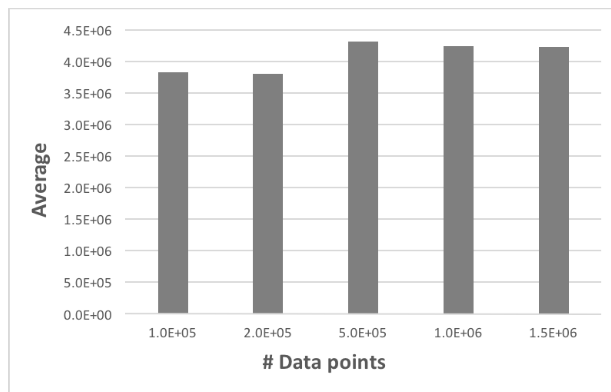
c) Extractor: This module was proven to be the bottleneck of Chimera. It has to deserialize the byte stream and unmarshal it into a domain object. The multi-threaded implementation was able to go up to 0.9M data points rebuilt per second: a high backpressure was experienced on the channels pushing data at the line rate, producing high GC overhead on long runs.

4) Processor Outbound Throughput:

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-553-1

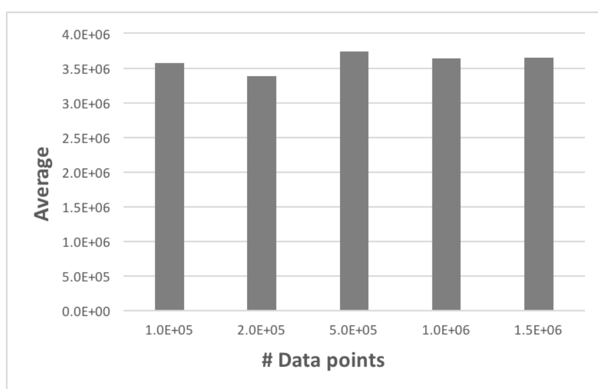


(a) Channel stress test results. Synthetic traffic was pulled from the ring buffer and pushed on the network, targeting the designated processor.

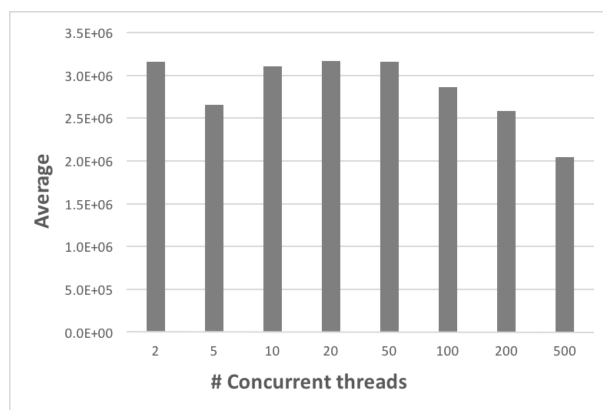


(b) Channel stress test results. Synthetic traffic was pulled from the ring buffer and pushed on the network, targeting the designated localhost processor.

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.

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 place was 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 I, 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 [31]. 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

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-553-1

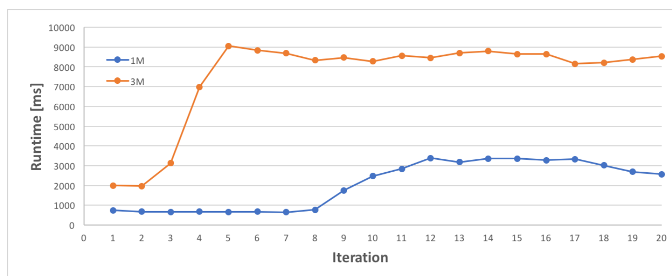


Figure 8. Experimented HDD-induced thrashing phenomenon. The I/O bottleneck put backpressure on the kernel, inducing high thrashing, which was impacting the overall functioning of the machine.

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 [32].

V. 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, start 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 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.

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 have 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 [33] 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 and one processor. 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.

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

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

[3] 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.

[4] 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.

[5] 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.

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

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

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

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

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

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

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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.

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

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

[24] 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.

[25] 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.

[26] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 4, no. 3, 1982, pp. 382–401.

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

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

[31] 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.

[32] L. Wirzenius and J. Oja, "The linux system administrators guide," versión 0.6, vol. 2, 1993.

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

Copyright (c) IARIA, 2017. ISBN: 978-1-61208-553-1