Twitter Heron: Stream Processing at Scale

Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg,
Sailesh Mittal, Jignesh M. Patel[*,1], Karthik Ramasamy, Siddarth Taneja

@sanjeevrk, @challenger_nik, @Louis_Fumaosong, @vikkyrk, @cckellogg,
@saileshmittal, @pateljm, @karthikz, @staneja

Twitter, Inc., *University of Wisconsin – Madison

ABSTRACT

Storm has long served as the main platform for real-time analytics at Twitter. However, as the scale of data being processed in real-time at Twitter has increased, along with an increase in the diversity and the number of use cases, many limitations of Storm have become apparent. We need a system that scales better, has better debug-ability, has better performance, and is easier to manage – all while working in a shared cluster infrastructure. We considered various alternatives to meet these needs, and in the end concluded that we needed to build a new real-time stream data processing system. This paper presents the design and implementation of this new system, called Heron. Heron is now the de facto stream data processing engine inside Twitter, and in this paper we also share our experiences from running Heron in production. In this paper, we also provide empirical evidence demonstrating the efficiency and scalability of Heron.

ACM Classification

H.2.4 [Information Systems]: Database Management—*systems*

Keywords

Stream data processing systems; real-time data processing.

1. INTRODUCTION

Twitter, like many other organizations, relies heavily on real-time streaming. For example, real-time streaming is used to compute the real-time active user counts (RTAC), and to measure the real-time engagement of users to tweets and advertisements. For many years, Storm [16, 20] was used as the real-time streaming engine inside Twitter. But, using Storm at our current scale was becoming increasingly challenging due to issues related to scalability, debug-ability, manageability, and efficient sharing of cluster resources with other data services.

A big challenge when working with Storm in production is the issue of debug-ability. When a topology misbehaves – which could be for a variety of reasons including load changes, misbehaving user code, or failing hardware – it is important to quickly determine the root-causes for the performance degradation. In Storm, work from multiple components of a topology is bundled into one operating system process, which makes debugging very challenging. Thus, we needed a cleaner mapping from the logical units of computation to each physical process. The importance of such clean mapping for debug-ability is really crucial when responding to pager alerts for a failing topology, especially if it is a topology that is critical to the underlying business model.

In addition, Storm needs dedicated cluster resources, which requires special hardware allocation to run Storm topologies. This approach leads to inefficiencies in using precious cluster resources, and also limits the ability to scale on demand. We needed the ability to work in a more flexible way with popular cluster scheduling software that allows *sharing* the cluster resources across different types of data processing systems (and not just a stream processing system). Internally at Twitter, this meant working with Aurora [1], as that is the dominant cluster management system in use.

With Storm, provisioning a new production topology requires manual isolation of machines, and conversely, when a topology is no longer needed, the machines allocated to serve that topology now have to be decommissioned. Managing machine provisioning in this way is cumbersome. Furthermore, we also wanted to be far more efficient than the Storm system in production, simply because at Twitter's scale, any improvement in performance translates into significant reduction in infrastructure costs and also significant improvements in the productivity of our end users.

We wanted to meet all the goals outlined above without forcing a rewrite of the large number of applications that have already been written for Storm; i.e. compatibility with the Storm and Summingbird APIs was essential. (Summingbird [8], which provides a Scala-idiomatic way for programmers to express their computation and constraints, generates many of the Storm topologies that are run in production.)

After examining various options, we concluded that we needed to design a new stream processing system to meet the design goals outlined above. This new system is called Heron. Heron is API-compatible with Storm, which makes it easy for Storm users to migrate to Heron. All production topologies inside Twitter now run on Heron. Besides providing us significant performance improvements and lower resource consumption over Storm, Heron also has big advantages in terms of debug-ability, scalability, and manageability.

In this paper, we present the design of Heron, and also present results from an empirical evaluation of Heron. We begin by briefly describing related work in the next section. Then, in Section 3, we describe Storm and motivate the need for Heron.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
ACM 978-1-4503-2758-9/15/05.
http://dx.doi.org/10.1145/2723372.2723374

[1] Work done while consulting for Twitter.

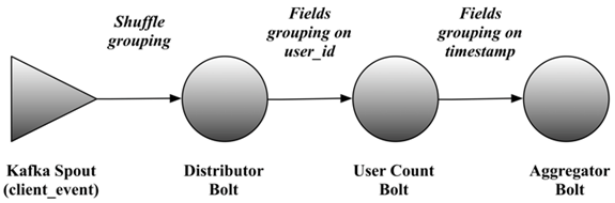

Figure 1: The RTAC Topology

Section 4 describes alternatives that we considered to address the problems described above, and Section 5 presents the design of Heron. Section 6 describes the tools around Heron that we use in production, and describes the current status of Heron inside Twitter. Results from an empirical evaluation comparing Storm and Heron are presented in Section 7. Finally, Section 8 contains our concluding remarks and points to some directions for future work.

2. Related Work

The interest in stream data processing systems includes a flurry of initial work about a decade ago (e.g. [6, 7, 10]). The need for highly-scalable stream processing systems has lead to the creation of a number of recent systems, including [2, 3, 5, 9, 15, 17, 18]. Stream processing has also been integrated with traditional database products (e.g. [4, 12, 19]), highlighting the need for stream processing in the broader enterprise ecosystem.

For our needs at Twitter, we needed a stream processing platform that was open-source, high-performance, scalable, and was compatible with the current Storm API. We also needed the platform to work on a shared infrastructure. These requirements limited the options discussed above. We did consider alternatives that came close to fitting our needs (see Section 4), but in the end concluded that we needed to build Heron.

3. Motivation for Heron

Storm has served the real-time analytics needs at Twitter for the past several years. Based on our operational experience at the current Twitter-scale, we identified several limitations with Storm that are highlighted in the following sections. These limitations motivated us to develop Heron.

3.1 Storm Background

As described in [20], a Storm topology is directed graph of *spouts* and *bolts*. Spouts are sources of input data (e.g. a stream of Tweets), and bolts are an abstraction to represent computation on the stream. Spouts often pull data from queues, such as Kafka [14] and Kestrel [13], and generate a stream of tuples, which is then fed into a network of bolts that carry out the required computation. For example, a topology that counts the number of active users in real-time (RTAC) is shown in Figure 1.

Spouts and bolts are run as *tasks*, and multiple such tasks are grouped into an *executor*. In turn, multiple executors are grouped into a *worker*, and each worker runs as a JVM process (as shown in Figure 2). A single host may run multiple worker processes, but each of them could belong to different topologies.

3.2 Storm Worker Architecture: Limitations

As described in [20] and briefly described above, a Storm worker has a fairly complex design. Several instances of worker processes are scheduled by the operating system in a host. Inside the JVM process, each executor is mapped to two threads. In turn, these

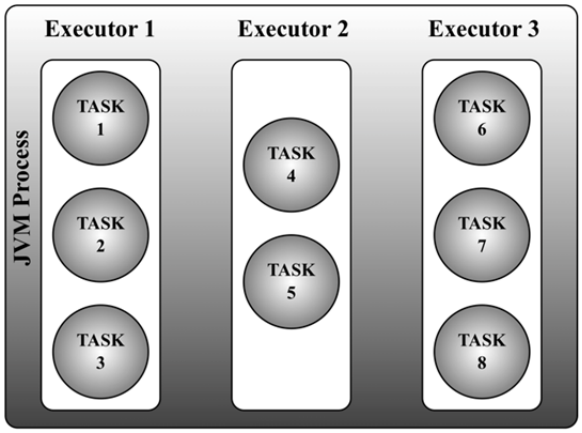

Figure 2: Storm Worker

threads are scheduled using a preemptive and priority-based scheduling algorithm by the JVM. Since each thread has to run several tasks, the executor implements another scheduling algorithm to invoke the appropriate task, based on the incoming data. Such multiple levels of scheduling and their complex interaction often leads to uncertainty about when the tasks are being scheduled.

Furthermore, each worker can run disparate tasks. For example, a Kafka spout, a bolt that joins the incoming tuples with a Twitter internal service, and another bolt writing output to a key-value store might be running in the same JVM. In such scenarios, it is difficult to reason about the behavior and the performance of a particular task, since it is not possible to isolate its resource usage. As a result, the favored troubleshooting mechanism is to restart the topology. After restart, it is perfectly possible that the misbehaving task could be scheduled with some other task(s), thereby making it hard to track down the root cause of the original problem.

Since logs from multiple tasks are written into a single file, it is hard to identify any errors or exceptions that are associated with a particular task. The situation gets worse quickly if some tasks log a larger amount of information compared to other tasks. Furthermore, an unhandled exception in a single task takes down the entire worker process, thereby killing other (perfectly fine) running tasks. Thus, errors in one part of the topology can indirectly impact the performance of other parts of the topology, leading to high variance in the overall performance. In addition, disparate tasks make garbage collection related-issues extremely hard to track down in practice.

For resource allocation purposes, Storm assumes that every worker is homogenous. This architectural assumption results in inefficient utilization of allocated resources, and often results in over-provisioning. For example, consider scheduling 3 spouts and 1 bolt on 2 workers. Assuming that the bolt and the spout tasks each need 10GB and 5GB of memory respectively, this topology needs to reserve a total of 15GB memory per worker since one of the worker has to run a bolt and a spout task. This allocation policy leads to a total of 30GB of memory for the topology, while only 25GB of memory is actually required; thus, wasting 5GB of memory resource. This problem gets worse with increasing number of diverse components being packed into a worker, and this situation happens frequently when generating (complex) topologies using higher-level abstractions like Summingbird [8].

As a consequence of allocating large memory to the workers, the use of common profiling tools, such as *jstack* or *heap dump*, becomes very cumbersome. When a worker is executing a heap dump, it misses sending heartbeats signals (which are needed to keep the worker alive), leading the Storm supervisor to kill it, thereby preventing the heap dump. Debugging problems becomes quite challenging as a result of this behavior.

A natural question is whether we could have re-architected Storm to allow it to run one task per worker. We considered this option, but discovered that approach could lead to big inefficiency in resource usage, and also limit the degree of parallelism that we could achieve. Such a deployment would result in a large number of workers per topology. Due to the homogeneity assumption across workers, there could be significant overprovisioning of resources. Under this model, we would have to reserve the following amount of memory for each worker:

$$SUM(Component_{Parallelism}) \times MAX\left(Component_{Resource_{Usage}}\right)$$

This number may be far larger than the optimal/ideal utilization. Referring to the aforementioned example with 3 spouts and 1 bolt, each worker needs around 10GB of memory, requiring a total of 40GB of memory, compared to the optimal memory size of 25GB. Finally, as the degree of parallelism for each component (bolts/tasks) increases, every worker tends to be connected to every other worker, which can run into problems with not having enough number of ports in each worker for communication, and thereby reduces the scalability options.

Storm workers use several threads and queues to move data between tasks and workers (see [20] for more details). A global receive thread in each worker process is responsible for getting data from workers "upstream", and a global send thread is in charge of transmitting data to the workers "downstream". In addition to these global threads, each executor consists of a *user logic thread* that runs the topology code, and a local *send thread* that communicates the output data from the user logic thread to the global sender thread. Hence in Storm, each tuple has to pass through four threads from the point of entry to the point of exit inside the worker process[2]. This design leads to significant overhead and queue contention issues.

3.3 Issues with the Storm Nimbus

The Storm Nimbus [20] performs several functions including scheduling, monitoring, and distributing JARs. It also serves the metrics-reporting component for the system, and manages counters for several topologies. Thus, the Nimbus component is functionally overloaded, and often becomes an operational bottleneck for a variety of reasons, as outlined below.

First, the Nimbus scheduler does not support resource reservation and isolation at a granular level for Storm workers. Consequently, Storm workers that belong to different topologies running on the same machine could interfere with each other. This situation can in turn lead to untraceable performance issues. To mitigate this problem, we ran production Storm topologies in isolation; i.e. entire machines are dedicated to a topology. But, this approach leads to wastage of resources, as it is hard for a topology to fully use all the hardware resources that are allocated to it (and all the time). Attempts to address this issue by running Storm on YARN [22] don't fully solve the problem.

Second, as mentioned in [20], Storm uses Zookeeper extensively to manage heartbeats from the workers and the supervisors. This use of Zookeeper limits the number of workers per topology, and the total number of topologies in a cluster, as at very large numbers, Zookeeper becomes the bottleneck. To address this issue, we had developed an interim design to route the keep-alive heartbeat traffic to special "heartbeat" daemons that ran in a separate set of machines. However, this interim design increased the operational burden, requiring separate monitoring of those hosts and the heartbeat daemons.

Finally, the Nimbus component is a single point of failure. When the Nimbus fails, the users are neither able to submit any new topologies nor kill existing ones. Furthermore, when Nimbus fails, any existing topology that undergoes failures cannot be automatically detected and recovered.

3.4 Lack of Backpressure

Storm has no backpressure mechanism. If the receiver component is unable to handle incoming data/tuples, then the sender simply drops tuples. This is a fail-fast mechanism, and a simple strategy, but it has the following disadvantages:

- If acknowledgements are disabled, this mechanism will result in unbounded tuple drops, making it hard to get visibility about these drops.
- Work done by upstream components is lost.
- System behavior becomes less predictable.

In extreme scenarios, this design causes the topology to not make any progress while consuming all its resources.

3.5 Efficiency

In production, there were several instances of unpredictable performance during topology execution, which then lead to tuple failures, tuple replays, and execution lag (rate of data arrival exceeds rate of processing by the topology). The most common causes for these reduced performance scenarios were:

- **Suboptimal replays** – A tuple failure anywhere in the tuple tree leads to failure of the entire tuple tree. This effect is more pronounced with high fan-out topologies where the topology is not doing any useful work, but is simply replaying the tuples.
- **Long Garbage Collection cycles** – Topologies consuming large amount of RAM for a worker encounter garbage collection (GC) cycles greater than a minute, resulting in high latencies and high tuple failure rates.
- **Queue contention** – In some cases, there is a lot of contention at the transfer queues, especially when a worker runs several executors.

To mitigate the risks associated with these issues, we often had to overprovision the allocated resources. Such overprovisioning has obvious negative implications on the infrastructure costs.

For example, a sample three-stage Storm topology (constructed primarily for this evaluation) requires approximately 600 cores with an average CPU utilization of 20-30%. To better understand where the time might be spent in such a topology, a simple Java program was written to absorb all such tuples and de-serializing them using Thrift. This step of processing input data consumed only 25 cores at 90% utilization. This resource consumption is equivalent to 75 cores at 30% CPU utilization, which is 8X lower than the 600 cores.

Even in the worst case, assuming that the counting and data movement overhead is as great as deserialization and reading from the source, one would expect the topology to use 150 cores. However, this topology runs on 600 cores, which indicates that

[2] This feature is implemented using fast disruptor queues and 0mq [23].

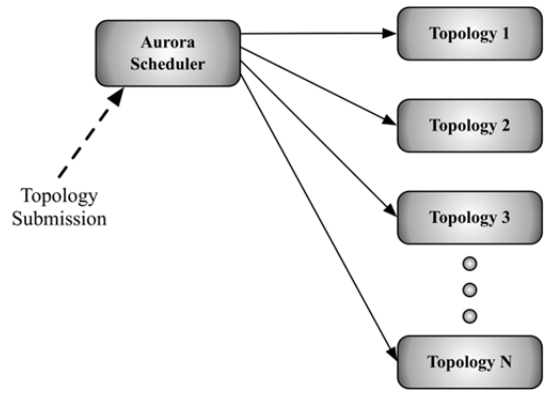

Figure 3: Heron Architecture

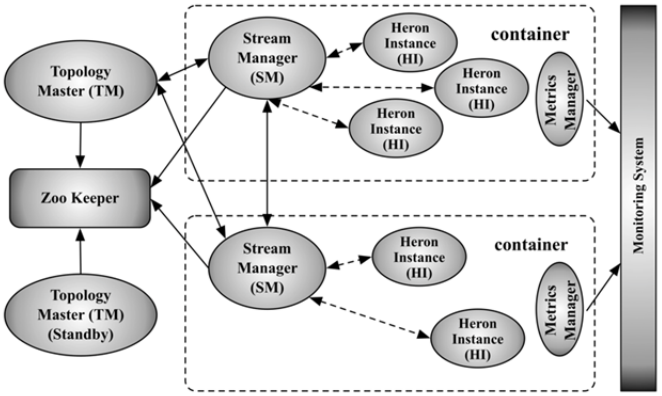

Figure 4: Heron Topology Architecture

with a better architecture/system we have the potential to achieve significant savings in resources that are consumed when running Storm topologies.

4. Design Alternatives

Considering the aforementioned issues, we weighed the options of whether to extend Storm, or to use another existing system, or to write a new system.

Since the issues discussed in Section 3 are fundamental to Storm, fixing them in Storm would have required extensive rewrite of the core components. At a high level, Storm organizes topologies as a bunch of queues that move data around, and changing this basic architectural block is hard. Modifying this existing system in such a fundamental way would have been inflexible, and potentially required much longer development cycles.

The next option was to consider using another existing open-source solution, such as Apache Samza [2] or Spark Streaming [18]. However, there are a number of issues with respect to making these systems work in its current form at our scale. In addition, these systems are not compatible with Storm's API. Rewriting the existing topologies with a different API would have been time consuming resulting in a very long migration process. Also note that there are different libraries that have been developed on top of the Storm API, such as Summingbird [8], and if we changed the underlying API of the streaming platform, we would have to change other components in our stack.

Thus, we concluded that our best option was to rewrite the system from ground-up, reusing and building on some of the existing components within Twitter.

5. Heron

In this section, we briefly describe the Heron data model, API, and the various Heron components.

5.1 Data Model and API

A key design goal for Heron is to maintain compatibility with the Storm API. Thus, the data model and API for Heron are identical to that of Storm [20]. Like Storm, Heron runs *topologies*. A topology is a directed acyclic graph of spouts and bolts. Like Storm, spouts generate the input tuples that are fed into the topology, and bolts do the actual computation.

A Heron topology is equivalent to a logical query plan in a database system. Such a logical plan is translated into a physical plan before actual execution. As a part of the topology, a programmer specifies the number of tasks for each spout and each bolt (i.e. the degree of parallelism), and how the data is partitioned as it moves across the spout and the bolt tasks (grouping). The actual topology, parallelism specification for each component, and the grouping specification, constitute the physical execution plan that is executed on the machines.

Finally, Heron's tuple processing semantics are similar to that of Storm, and include the following:

- *At most once* – No tuple is processed more than once, although some tuples may be dropped, and thus may miss being analyzed by the topology.
- *At least once* – Each tuple is guaranteed to be processed at least once, although some tuples may be processed more than once, and may contribute to the result of the topology multiple times.

5.2 Architecture overview

Since the key factors driving the need for Heron are to ease the task of manageability, improve developer productivity, and improve the predictability of performance, we had to make careful decisions about how to architect the different components of the system considering clean abstractions for various interconnected modules, and ensuring an architecture that can operate at Twitter's scale.

The overall architecture for Heron is shown in Figure 3. Users employ the Heron (spouts/bolts programming) API to create and deploy topologies to the *Aurora scheduler*, using a Heron command line tool. Aurora [1] is a generic service scheduler that runs as a framework on top of Mesos [11]. However, our architecture implements a scheduler abstraction that facilitates running Heron on other schedulers such as YARN, Mesos, and ECS (Amazon EC2 Docker Container Service). This design is a departure from Storm, where Nimbus (which is an integral component of Storm) was used for scheduling. Since Twitter's homegrown Aurora scheduler and other open-source schedulers (e.g. YARN) have become sophisticated, we made the conscious choice of working with these schedulers rather than implementing another one.

Each topology is run as an Aurora job consisting of several *containers*, as shown in Figure 4. The first container runs a process called the *Topology Master*. The remaining containers each run a *Stream Manager*, a *Metrics Manager*, and a number of processes called *Heron Instances* (which are spouts/bolts that run user logic code). Multiple containers can be launched on a single

physical node. These containers are allocated and scheduled by Aurora based on the resource availability across the nodes in the cluster. (At Twitter, Aurora maps these containers to Linux *cgroups*.) A standby Topology Master can be run for availability. The metadata for the topology (which includes information about the user who launched the job, the time of launch, and the execution details) are kept in Zookeeper.

The Heron Instances are written in Java, as they need to run user logic code (which is written in Java). There is one JVM per Heron Instance.

All Heron processes communicate with each other using protocol buffers (*protobufs*).

5.3 Topology Master

The *Topology Master* (**TM**) is responsible for managing the topology throughout its existence. It provides a single point of contact for discovering the status of the topology (and thus is similar to the Application Master in YARN). Upon startup, the TM makes itself discoverable by creating an ephemeral node at a well-known location in Zookeeper. The ephemeral node serves the following two purposes:

- It prevents multiple TMs from becoming the master for the same topology, thereby providing different processes of the topology a consistent view of the entire topology.
- It allows any other process that belongs to the topology to discover the TM.

The TM also serves as a gateway for the topology metrics through an endpoint. Note that since the TM is not involved in the data processing path, it is not a bottleneck.

5.4 Stream Manager

The key function of the *Stream Manager* (**SM**) is to manage the routing of tuples efficiently. Each *Heron Instance* (**HI**) connects to its local SM to send and receive tuples. All the SMs in a topology connect between themselves to form a $O(\boldsymbol{k}^2)$ connection network, where $\boldsymbol{k}$ is the number of containers/SMs in the physical plan of the topology. Note that since the number of HIs, $\boldsymbol{n}$ is generally much larger than $\boldsymbol{k}$, this design permits a way to scale the communication overlay network by multiplexing $O(n^2)$ logical channels over $O(k^2)$ physical connections. Furthermore, any tuples routed from one HI to another HI in the same container is routed using a local short-circuiting mechanism.

5.4.1 Topology Backpressure

Unlike Storm, Heron employs a *backpressure* mechanism to dynamically adjust the rate at which data flows through the topology. This mechanism is important in topologies where different components can execute at different speeds (and the speed of processing in each component can change over time). For example, consider a pipeline of work in which the later/downstream stages are running slow, or have slowed down due to data or execution skew. In this case, if the earlier/upstream stages do not slow down, it will lead to buffers building up long queues, or result in the system dropping tuples. If tuples are dropped mid-stream, then there is a potential loss in efficiency as the computation already incurred for those tuples is wasted. A backpressure mechanism is needed to slow down the earlier stages. We considered a few implementation strategies, which we describe next.

TCP Backpressure: In this strategy, we use the TCP windowing mechanism to propagate backpressure from the HIs to the other upstream components. Since the HIs and the SM (in each container) communicate using TCP sockets, the rate of draining from the send/receive buffers is equal to the rate of production/consumption by the local HI. If an HI is executing slowly, then its receive buffer will start filling up. The SM that is pushing data to this HI will recognize this situation, as its send buffer will also fill up. This backpressure mechanism then propagates to the other SMs and HIs upstream. Note that this backpressure is only cleared when the original (slow) HI starts catching up again.

This simple TCP-based backpressure mechanism is easy to implement. However, this method did not work well in practice because multiple logical channels (between HIs) are overlaid on top of the physical connections between SMs. This multiplexing inadvertently not only causes the upstream HIs to slow down, but also often causes the downstream HIs (that are on the same connection) to also slow down. Consequently, any congestion clears very slowly, causing the entire topology to experience significant and unduly long-lasting performance degradation.

Spout Backpressure: In this approach, the SMs clamp down their local spouts to reduce the new data that is injected into the topology. This approach is used in conjunction with TCP backpressure between the SMs and the HIs. When an SM realizes that one or more of its HIs are slowing down, it identifies its local spouts and stops reading data from them. This mechanism has the effect of slowing down the spout as the spout's send buffer that is used to send tuples to the SM will get filled up, and will eventually block. The affected SM sends a special **start backpressure** message to other SMs requesting them to clamp down their local spouts. When the other SMs receive this special message, they oblige by not reading tuples from their local spouts. Once the slow HI catches up, the local SM sends **stop backpressure** messages to other SMs. When the other SMs receive this special message, they restart consuming data from their local spouts again.

This approach directly clamps down the most upstream component (spouts). This method may be less than optimal because we may unnecessarily clamp down a spout, when simply slowing down an immediate upstream producer is all that was actually necessary. The other potential disadvantage of this approach is the additional message passing overhead. However, the advantage of this approach is that the reaction time to flow rate changes is small, irrespective of the depth of the topology.

Stage-by-Stage Backpressure: A topology can be viewed as consisting of multiple stages. In this approach, we gradually propagate the backpressure stage-by-stage until it reaches the spouts (which represent the 1st stage in any topology). As in the spout backpressure method, this strategy is used in conjunction with the TCP backpressure mechanism between the SMs and the HIs, and differs in the backpressure control messages that are exchanged between the SMs.

5.4.2 Implementation

In Heron we have implemented the spout backpressure approach, as it is simpler to implement. This mechanism works well in practice, and also aids debug-ability as one can see when skew-related events happen, and which component was the root cause of the backpressure trigger.

Every socket channel is associated with an application-level buffer that is bounded in size by both a high water mark and a low water mark. Backpressure is triggered when the buffer size reaches the high water mark, and remains in effect until the buffer

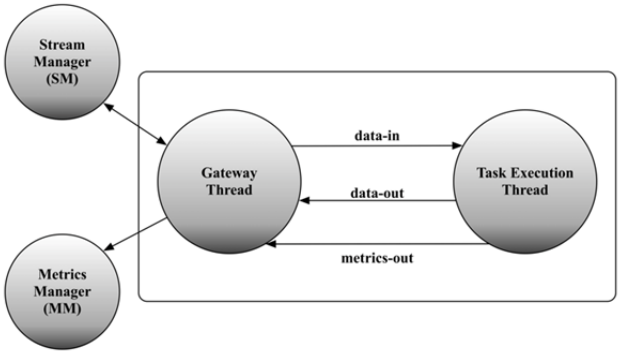

Figure 5: Heron Instance

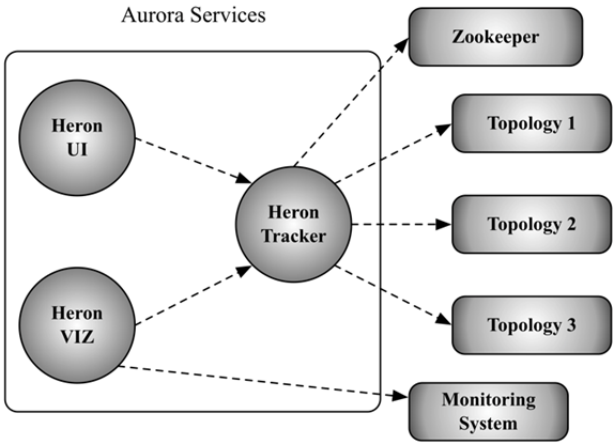

Figure 6: Heron Services for Production

size goes below the low water mark. The rationale for this design is to prevent a topology from rapidly oscillating between going into and coming out of the backpressure mitigation mode.

A consequence of this design is that once a tuple is emitted from the spout, Heron does not drop it, except during process or machine failure scenarios. This behavior makes tuple failures more deterministic.

When a topology is in backpressure mode, it goes as fast as the slowest component. If this situation continues to persist for a while, it could lead to data building up in the "source" queues from which the spout reads the data. Depending on the topology, spouts can be configured to drop older data.

5.5 Heron Instance

The main work for a spout or a bolt is carried out in the *Heron instances* (**HI**s). Unlike the Storm worker, each HI is a JVM process, which runs only a single task of the spout or the bolt. Such a design allows us to easily debug/profile either a spout or bolt, since the developer can easily see the sequence of events and logs that originate from a HI.

Note that since the complexity of data movement has been moved to the SMs, it easy for us to consider writing native HIs in other languages in the future.

To implement the HIs, we considered two designs: one using a single thread and the other using two threads. Next, we describe these two designs.

5.5.1 Single-threaded approach

In the single-threaded design, a main thread maintains a TCP communication channel to the local SM and waits for tuples. Once a tuple arrives, the user logic code is invoked in the same thread. If the user logic code program generates an output tuple, it is buffered. Once the buffer exceeds a certain threshold, it is delivered to the local SM.

While this approach has the advantage of simplicity, it has several disadvantages, as the user code can potentially block due to a variety of reasons, including:

- Invoking the sleep system call for a finite duration of time
- Using read/write system calls for file or socket I/O
- Calling thread synchronization primitives

We implemented this approach and realized that such blocking is not desirable for the required periodic activities such as metrics reporting. Since the duration of blocking could potentially vary, it leads to unpredictable behavior. If the metrics are not collected and sent timely, one cannot reliably troubleshoot whether an HI is in a "bad" state.

5.5.2 Two-threaded approach

In this design, the HIs have two threads namely, a *Gateway thread* and a *Task Execution thread* as shown in Figure 5. The Gateway thread is responsible for controlling all the communication and data movement in and out from the HI. It maintains TCP connections to the local SM and the metrics manager. It is also responsible for receiving incoming tuples from the local SM. These tuples are sent to the Task Execution thread for processing.

The Task Execution thread runs user code. When the task execution thread is started, it executes the "open" or "prepare" method depending upon whether the instance is executing a spout or a bolt, respectively. In the case of a bolt, when tuples arrive, the task execution thread invokes the "execute" method with the incoming tuple for processing. In the case of a spout, it repeatedly calls the "nextTuple" method to fetch data from the source, and then injects this data as tuples into the topology. The emitted tuples from either spout or bolt are sent to the Gateway thread, which forwards the tuples to the local SM. In addition to processing tuples, the Task Execution thread collects various metrics such as the number of tuples executed, the number of tuples emitted, the number of tuples acknowledged, and the latency experienced during the processing of tuples.

The Gateway thread and the Task Execution thread communicate between themselves using three unidirectional queues, as shown in Figure 5. The Gateway thread uses the *data-in* queue to push tuples to the Task Execution thread for processing. The Task Execution thread uses the *data-out* queue to send tuples to the Gateway thread (for sending to other parts of the topology). The *metrics-out* queue is used by the Task Execution thread to pass the collected metrics to the Gateway thread.

The data-in and the data-out queues are bounded in size. The Gateway Execution thread stops reading from the local SM when the data-in queue exceeds this bound. This action triggers the backpressure mechanism at the local SM. Similarly, when items in the data-out queue exceed the bound, the Gateway thread can

assume that the local SM can not receive more data, and that the Task Execution thread should not emit or execute any more tuples.

When we ran large topologies in production with bounded queue sizes, we often experienced unexpected GC issues. Everything worked fine until, say, a network outage happened and the Gateway thread was unable to send tuples from the data-out queue. Tuples would then start to back up in the data-out queue, and because they are live objects, they could not be reclaimed. This situation then caused the corresponding HI to reach its memory limit. Once the network recovered, the Gateway thread would start reading tuples from the local SM, as well as sending out tuples from data-out queue. If the Gateway thread read tuples from the SM before sending out tuples, any new object construction could trigger the GC, since nearly all of the available memory was already used up, quickly resulting in further performance degradation.

To avoid such GC issues, we periodically check the capacity of the data-out and data-in queues and increase/decrease these queue sizes accordingly. If the capacity of the queue grows over a configurable limit, then it is reduced (currently to half of the last capacity). This mechanism is invoked periodically until the capacity of the queue returns to a stable constant value, or the capacity reaches zero. When the capacity of the queue becomes zero, neither new tuples can be injected, nor, in many cases, can new tuples be produced. As a consequence, it is easier to recover from GC issues. Similarly, when the outstanding number of tuples in the queue is smaller than the configured limit, the capacity is gradually increased until the queue length either reaches the configured limit or hits the maximum capacity value.

5.6 Metrics Manager

The *Metrics Manager* (**MM**) collects and exports metrics from all the components in the system. These metrics include system metrics and user metrics for the topologies. There is one metrics manager for each container, to which the Stream Manager and Heron Instances report their metrics.

Metrics are sent from each container to an in-house monitoring system. The MMs also pass the metrics to the Topology Master for displaying in external UIs. The separation of metrics reporting using local MM provides us the flexibility to support other monitoring systems (such as Ganglia and Graphite) in the future.

5.7 Startup Sequence and Failure Scenarios

When a topology is submitted to Heron, a sequence of steps are triggered. Upon submission, the scheduler (in our case it is generally Aurora [1]) allocates the necessary resources and schedules the topology containers in several machines in the cluster. The Topology Master (TM) comes up on the first container, and makes itself discoverable using the Zookeeper ephemeral node. Meanwhile, the Stream Manager (SM) on each container consults Zookeeper to discover the TM. The SM then connects to the TM and periodically sends heartbeats.

When all the SMs are connected, the TM runs an assignment algorithm to assign different components of the topology (spouts and bolts) to different containers. This is called the *physical plan* in our terminology. Once the assignment is complete, the SMs get the entire physical plan from the TM, which helps the SMs to discover each other. Now the SMs connect to each other to form a fully-connected network. Meanwhile, the Heron instances (HI) come up, discover their local SM, download their portion of the physical plan, and start executing. After these steps are completed, data/tuples starts flowing through the topology. For safekeeping, the TM writes the physical plan to Zookeeper to rediscover the state in case of its failure.

When a topology is executing, there are several failure scenarios that could affect some portion of the topology, and sometimes even the entire topology itself. These scenarios consist of the death of processes, failure of containers, and failures of machines.

When the TM process dies, the container restarts the failed process, and the TM recovers its state from Zookeeper. When a topology is started with a standby TM, the standby TM becomes the master, and the restarted TM becomes the standby. Meanwhile, the SMs that have open channels to the TM rediscover the new TM, and connect to it.

Similarly when an SM dies, it gets restarted in the same container, it rediscovers the TM, and it initiates a connection to fetch the physical plan to check if there are any changes in its state. Other SMs, who have lost the connection to the failed SM, also get a copy of the same physical plan indicating the location of the new SM, and create a connection to the new SM. When an instance (HI) dies within a container, it is restarted, and it contacts its local SM. The HI then gets a copy of the physical plan, identifies whether it is a spout or bolt, and starts executing the corresponding user logic code.

When any container is rescheduled or relocated to a new machine, the newly minted SM discovers the TM, and follows the same sequence of steps of an SM failure and an HI failure.

5.8 Architecture Features: Summary

We note several salient aspects of our design. First, the provisioning of resources (e.g. for containers and even the Topology Master) is cleanly abstracted from the duties of the cluster manager, thereby allowing Heron to "play nice" with the rest of the (shared) infrastructure.

Second, since each Heron Instance is executing only a single task (e.g. running a spout or bolt), it is easy to debug that instance by simply using tools like *jstack* and *heap dump* with that process.

Third, the design makes it transparent as to which component of the topology is failing or slowing down, as the metrics collection is granular, and lets us easily map an issue unambiguously to a specific process in the system.

Fourth, by allowing component-level resource allocation, Heron allows a topology writer to specify exactly the resources for each component, thereby avoiding unnecessary over-provisioning.

Fifth, having a Topology Master per topology allows each topology to be managed independently of each other (and other systems in the underlying cluster). In additional, failure of one topology (which can happen as user-defined code often gets run in the bolts) does not impact the other topologies.

Sixth, the backpressure mechanism allows us to achieve a consistent rate of delivering results, and a precise way to reason about the system. It is also a key mechanism that allows migrating topologies from one set of containers to another (e.g. to an upgraded set of machines).

Finally, we now do not have any single point of failure.

6. Heron in Production

To get Heron working in production, we needed several additional functionalities, which include: a) the ability for users to interact with their topologies, b) the ability for users to view metrics and

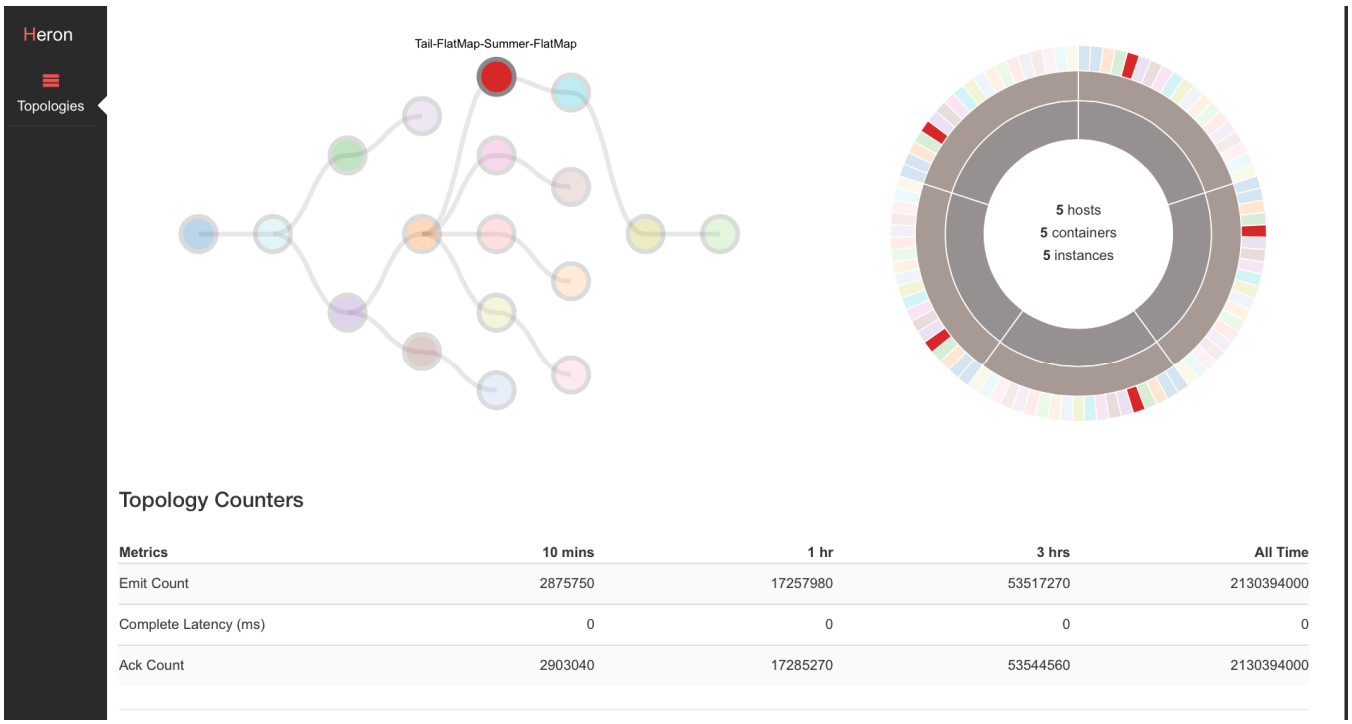

Figure 7: Topology Visualization. *The figure on the left shows the logical plan for the topology, and the figure on the right shows the current location of components of the physical plan for the topology. Clicking on a logical plan component (e.g. the red bolt in the figure on the left) highlights the location of the containers on the map on the right. The table below the two figures shows key metrics.*

trends for their topologies, c) the ability for users to view exceptions that occurs in the Heron Instances, and d) the ability for users to view their topology logs. To accommodate all this functionality, we implemented additional components: *Heron Tracker*, *Heron UI* and *Heron Viz* as shown in Figure 6. These components are described in more details below.

6.1 Heron Tracker

The Heron Tracker acts as a gateway to access several information about topologies. It interfaces with the same Zookeeper instance that the topologies use to save their metadata, and collects additional information about the topologies. The tracker uses Zookeeper watches to keep track of new topologies that are being launched, existing topologies that are being killed, and any change in the physical plan of the topology (such as a container being moved from one host to another). In addition to this information, the tracker also uses the metadata information in Zookeeper to discover where the Topology Master of a topology is running to obtain metrics and any other relevant data.

The tracker provides a clean abstraction by exposing a well-defined REST API that makes it easy to create any additional tools. The API provides information about the topologies such as the logical and the physical plan, various metrics including user-defined and system metrics, links to log files for all the instances, and links to the Aurora job pages for the executing containers. The tracker runs as an Aurora service, and typically is run in several instances for fault tolerance. The API requests are load balanced across these instances.

6.2 Heron UI

Heron users can interact with their topologies using a rich visual UI. This UI uses the Heron Tracker API and displays a visual representation of the topologies, including its logical and physical plan. Logical plan displays the directed acyclic graph with each node uniquely color-coded. The physical plan is displayed as a set of concentric circles, with the inner circle representing the hosts, the middle circle depicting the containers, and the outer circle representing the instances of components. A user can drill down either on a component or on an instance of the component to display metrics such as emit counts, complete/execute latencies, acknowledged counts, and fail counts for the time intervals of last 10 minutes, 1 hour, 3 hours, and since the start of the topology. In addition to these features, the UI also offers easy access links to view the logs and exceptions that are associated with an instance – an important feature for debugging. Figure 7 shows a part of the visualization for a 5-stage topology.

6.3 Heron Viz

Heron Viz is a service that creates the dashboard used to view the metrics collected by the Metrics Manager for a topology. This service periodically contacts the Heron Tracker for any new topology. When there is a new topology, it uses the HTTP API of graphing system, called Viz, to create a dashboard of graphs. In order to create the dashboard, Heron Viz retrieves the logical plan of the topology to determine the components, i.e. the bolts and the spouts, and the number of instances of each component. For each component in the topology, a section is created based on the type of the component (spout or bolt), and queries are generated based on the number of instances.

Broadly, the Heron Viz dashboard for a topology is categorized into health metrics, resource metrics, component metrics and stream manager (SM) metrics.

Health metrics include the overall lag the topology is experiencing, aggregate tuple fail count in the spouts, and number of SM deaths.

Figure 8: Topology Metrics Reporting

Resource metrics consist of the CPU resources allocated, CPU resources that are actually used, the amount of memory that is being used, the amount of memory that has been reserved, and the amount of time spent in GC.

Component metrics include, for each spout, the number of tuples that have been emitted, failed, and acknowledged. They also include the average end-to-end latency for processing a tuple. Additional component metrics include, for each bolt, the number of tuples executed, acknowledged and emitted, and the average latency for processing each tuple.

Finally, the SM metrics tracks for each SM, the number of tuples that have arrived from instances, the number of tuples delivered to the instances, the number of tuples dropped when receiving and sending to instances and other SMs, and the total aggregate time spent in backpressure mode.

A sample and partial view of the dashboard is shown in Figure 8.

6.4 Heron@Twitter

At Twitter, Storm has been decommissioned and Heron is now the de-facto streaming system. It has been in production for several months and runs hundreds of development and production topologies in multiple data centers. These topologies process several tens of terabytes of data, generating billions of output tuples.

Topologies vary in their complexity and a large number of topologies have three or fewer stages. There are several topologies that extend to more than three stages, and the longest ones go as deep as eight stages.

The use cases for these topologies are varied and include data transformation, filtering, joining, and aggregating content across various streams in Twitter (e.g. computing counts). The use cases also include running complex machine learning algorithms (e.g. regression, association and clustering) over streaming data. Various groups inside Twitter use Heron. These groups include user services, revenue, growth, search, and content discovery.

After migrating all the topologies to Heron (from Storm), there was an overall 3X reduction in hardware – a significant improvement in the infrastructure efficiency at Twitter's scale.

7. Empirical Evaluation

In this section, we present results comparing Heron and Storm.

7.1 Workload

We chose to evaluate Heron in the context of two topologies – a Word Count topology, and a RTAC topology (cf. Figure 1). For each topology, we considered two variants, one with acknowledgements enabled (i.e. at least once semantics), and the other with no acknowledgements (i.e. at most once semantics).

Note that both topologies were constructed primarily for this empirical evaluation, and should not be construed as being the representative topology for Heron/Storm workloads at Twitter.

7.2 Setup

All experiments were run on machines with dual Intel Xeon E5645@2.4GHZ CPUs, each consisting of 12 physical cores with hyper-threading enabled, 72GB of main memory, and 500GB of disk space. We tuned both Storm and Heron to perform in ways that we expect in production settings. In other words, there are no out-of-memory (OOM) crashes (or any other failure due to resource starvation during scheduling), or long repetitive GC cycles. The Storm topologies were run in isolation, which means that no process besides the kernel, Mesos slaves, and metric exporter daemons, is running in the system. Heron was running in a shared cluster, with Linux "cgroups" isolation.

The experiments were allowed to run for several hours to attain steady state before measurements were taken. For Storm, this means very small number of drops in the 0mq layer, and that the size of various queues are not growing, and remain small. For Heron, this means no backpressure, and that its transfer queues also maintain a stable size while remaining small.

Note that for topologies with acknowledgements, tuple failures may occur due to 0mq drops in Storm, or due to timeout. While

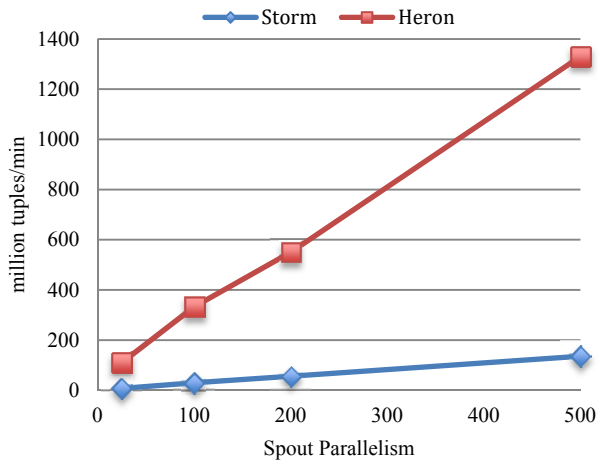

Figure 9: Throughput with acknowledgements

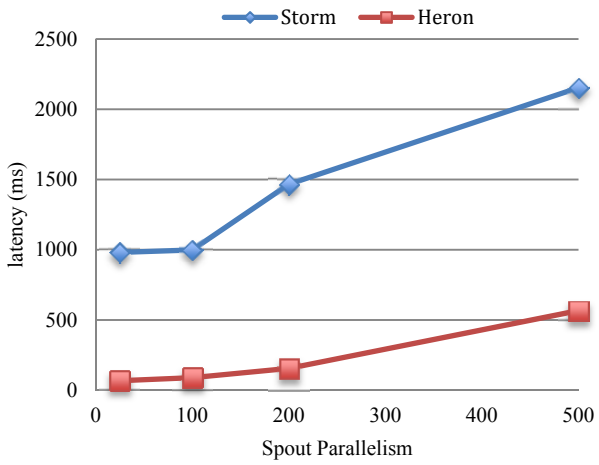

Figure 10: End-to-end latency with acknowledgements

for Heron, tuple failures can happen only due to timeouts. We used 30 seconds as the timeout interval in both cases.

7.3 Word Count Topology

In these set of experiments, we used a simple word count topology. In this topology, the spout tasks generate a set of random words (~175k words) during the initial "open" call, and during every "nextTuple" call. In each call, each spout simply picks a word at random and emits it. Hence spouts are extremely fast, if left unrestricted. Spouts use a fields grouping for their output, and each spout could send tuples to every other bolt in the topology.

Bolts maintain an in-memory map, which is keyed by the word emitted by the spout and updates the count when it receives a tuple.

This topology is a good measure of the overhead introduced by either Storm or Heron since it does not do significant work in its spouts and bolts.

For each set of experiments, we varied the number of Storm spout/bolt tasks, Heron spout/bolt instances, Storm workers, and Heron containers as shown below in Table 1.

Table 1: Experimental setup for the Word Count topology

Components	Expt. #1	Expt. #2	Expt. #3	Expt. #4
Spout	25	100	200	500
Bolt	25	100	200	500
# Heron containers	25	100	200	500
# Storm workers	25	100	200	500

7.3.1 Acknowledgements Enabled

In these experiments, the word count topology is enabled to receive acknowledgements. We measured the topology throughput, end-to-end latency, and CPU usage, and plot these results in Figure 9, Figure 10, and Figure 11 respectively. Each of these figures has four points on each line, corresponding to the four experimental setup configurations that are shown in Table 1.

As shown in Figure 9, the topology throughput increases linearly for both Storm and Heron. However, for Heron, the throughput is 10-14X higher than that for Storm in all these experiments.

The end-to-end latency graph, plotted in Figure 10, shows that the latency increases far more gradually for Heron than it does for Storm. Heron latency is 5-15X lower than that of the Storm. There are many bottlenecks in Storm, as the tuples have to travel through multiple threads inside the worker and pass through multiple queues. (See Section 3.)

In Heron, there are several buffers that a tuple has to pass through as they are transported from one Heron Instance to another (via the SMs). Each buffer adds some latency since tuples are transported in batches. In normal cases, this latency is approximately 20ms, and one would expect the latency to be of the same value since the tuples in this topology have the same number of hops. However, in this topology, the latency increases as the number of containers increase. This increase is a result of the SMs becoming a bottleneck, as they need to maintain buffers for each connection to the other SMs, and it takes more time to consume data from more buffers. The tuples also live in these buffers for longer time given a constant input rate (only one spout instance per container).

Figure 11 shows the aggregate CPU resources that are utilized across the entire cluster that is used for this topology, as reported

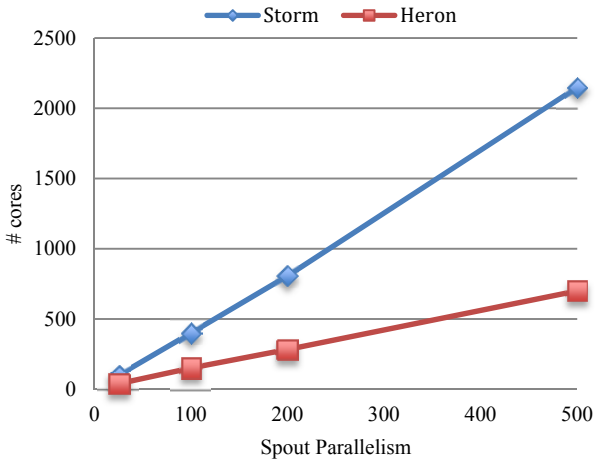

Figure 11: CPU usage with acknowledgements

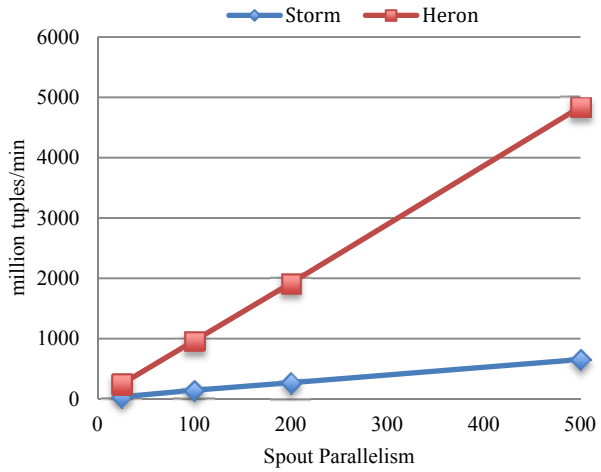

Figure 12: Throughput with acknowledgements disabled

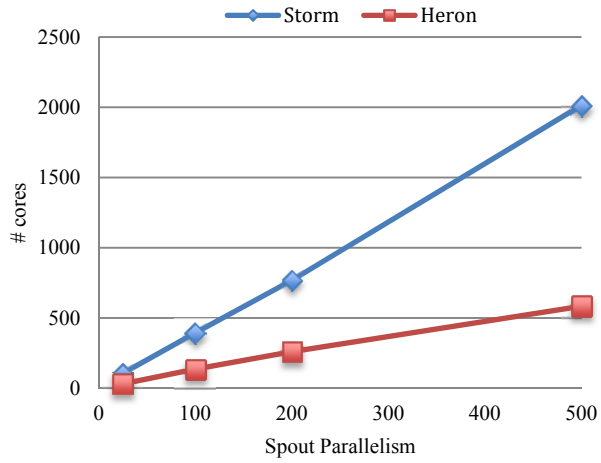

Figure 13: CPU usage with acknowledgements disabled

by Aurora. The metric in this figure is number of cores, and the aggregate CPU resources that is consumed is computed by taking the CPU utilization of each core that is used, and dividing it by 100.

As shown in Figure 11, the CPU usage also increases linearly as more data is pushed through the topology for both Storm and Heron. This behavior is expected as increasing the number of processes and the number of containers results in requiring more CPU resources. However, the CPU usage of Heron is 2-3X lower than that of the Storm, and the increase in CPU utilization is nearly linear as the number of containers increase.

7.3.2 Acknowledgements Disabled

In these experiments, we disable the acknowledgments, which means that some tuples might be lost mid-flight. We measured the throughput and the CPU usage, and show these results in Figure 12 and Figure 13, respectively.

As can be seen in Figure 12, the throughput increases linearly for both Storm and Heron as we input more data into the topology. However, across all the experiments, the throughput of Heron is 6-8X higher than that of Storm.

When comparing CPU usage (Figure 13), we observe that the CPU resources used by Heron is consistently 3-4X lower than that for Storm, while achieving a far higher throughput.

7.4 RTAC Topology

For this evaluation, we chose the example RTAC topology shown in Figure 1. We set up this topology so that the expected output rate for this topology, when it can keep up with the input data rate, is ~6M tuples/minute. Using iterative experiments, we identified the configuration parameters for Storm and Heron that provided the best performance. These configurations are listed in Table 2.

Table 2: Parameter settings for the RTAC topology

Component	# Storm tasks	# Heron tasks
Spout	200	60
DistributorBolt	200	15
UserCountBolt	300	3
AggregatorBolt	20	2
Workers/Containers	50	50

7.4.1 Acknowledgements Enabled

In the first experiment, we enable end-to-end acknowledgements in the topology. We measured the actual CPU usage and the end-to-end latency for the topology when running both in Storm and in Heron. Recall that Storm topologies run in isolation (cf. Section 3.3). The results plotting the total CPU resources utilized (in terms of 100% utilized core counts), and the end-to-end latencies are shown in Figure 14 and Figure 15, respectively.

As shown in these two figures, Storm needed 360 cores to keep up with the required throughput of 6M tuples/min, with an end-to-end tuple latency of 70ms. On the other hand, Heron can sustain the required throughput with just 36 cores, while delivering an end-to-end tuple latency of only 24ms. When we relaxed the latency requirements for Storm, we were able to sustain the required throughput with 240 cores with an increased end-to-end latency of 500ms. In this experiment, Heron shows *65-95% improvement in the latency over Storm, while requiring only 20-22% of the CPU resources that Storm requires*.

7.4.2 Acknowledgements Disabled

In the second experiment, we disabled acknowledgements, which means that failed tuples are dropped without any replaying. In this case, we measured the CPU usage. The results for this experiment are shown in Figure 16.

With this simpler topology, Storm needed 240 cores, creating an output throughput rate of 6M/min. On the other hand, Heron can keep up with this topology using just 20 cores, a *10X reduction in CPU resources* that are required.

8. Conclusions and Future work

The need for real-time stream analytics at Twitter continues to grow, and in production has pushed the boundaries of what existing streaming systems can deliver in terms of manageability and performance. To meet these needs, and to also provide backward compatibility with our existing streaming API, we have designed and implemented a new stream data processing system called Heron, which we have presented in this paper. We have also presented results from an empirical evaluation of Heron that demonstrates large reductions in CPU resources when using Heron, while delivering 6-14X improvements in throughput, and 5-10X reductions in tuple latencies.

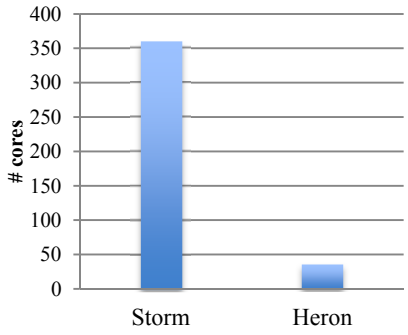

Figure 14: CPU usage with acknowledgements enabled

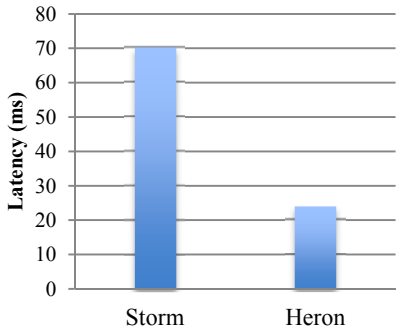

Figure 15: End-to-end latency with acknowledgements

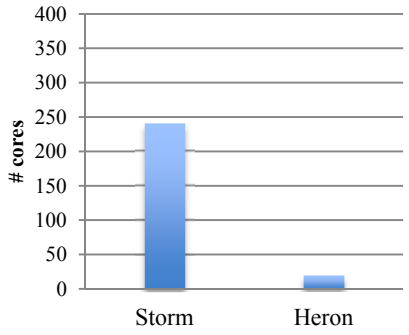

Figure 16: CPU usage with acknowledgements disabled

The design of Heron allows supporting *exactly once* semantics, but the first version of Heron does not have this implementation. One reason for tolerating the lack of exactly once semantics is that Summingbird [8] simultaneously generates both a Heron query and an equivalent Hadoop job, and in our infrastructure the answers from both these parts are eventually merged.

However, there is a real need for fast responses from the streaming system (even if the answer is not fully accurate) as this real-time analytics is crucial to how Twitter works. Exactly once semantics requires some form of check pointing (e.g. see [21]), which is known to reduce the performance, and our design allows for adding such semantics. We are considering designing and implementing mechanisms for exactly once semantics in Heron.

Acknowledgements

Replacing something as key as the streaming platform across the entire company would not have been possible without the help and active participation from many teams inside Twitter. Specifically we thank the Aurora team for their quick turnaround to our requests. The System Monitoring team deserves big kudos for making changes to handle the big set of metrics data that every Heron topology generates. We also thank the Summingbird team for working closely with us to make sure that Summingbird worked well with Heron. A special thanks to the Mesos team for working with us to design the scheduler abstraction. Finally, Heron would not have gotten into production without the Systems and Reliability Engineering team guiding us through every failure scenario to make our system more robust in production.

REFERENCES

[1] Apache Aurora. http://aurora.incubator.apache.org

[2] Apache Samza. http://samza.incubator.apache.org

[3] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, Sam Whittle: MillWheel: Fault-Tolerant Stream Processing at Internet Scale. *PVLDB 6(11)*: 1033-1044 (2013)

[4] Mohamed H. Ali, Badrish Chandramouli, Jonathan Goldstein, Roman Schindlauer: The Extensibility Framework in Microsoft StreamInsight. *ICDE* 2011: 1242-1253

[5] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, Shivakumar Venkataraman: Photon: Fault-tolerant and Scalable Joining of Continuous Data Streams. *SIGMOD* 2013: 577-588

[6] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, Jennifer Widom: STREAM: The Stanford Stream Data Manager. IEEE Data Eng. Bull. 26(1): 19-26 (2003)

[7] Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Eduardo F. Galvez, Jon Salz, Michael Stonebraker, Nesime Tatbul, Richard Tibbetts, Stanley B. Zdonik: Retrospective on Aurora. VLDB J. 13(4): 370-383 (2004)

[8] P. Oscar Boykin, Sam Ritchie, Ian O'Connell, Jimmy Lin: Summingbird: A Framework for Integrating Batch and Online MapReduce Computations. PVLDB 7(13): 1441-1451 (2014)

[9] DataTorrent. https://www.datatorrent.com

[10] Minos N. Garofalakis, Johannes Gehrke: Querying and Mining Data Streams: You Only Get One Look. VLDB 2002

[11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, Ion Stoica: Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. NSDI 2011

[12] IBM Infosphere Streams. http://www-03.ibm.com/software/products/en/infosphere-streams/

[13] Kestrel: A Simple, Sistributed Message Queue System. http://robey.github.com/kestrel

[14] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: A Distributed Messaging System for Log Processing. SIGMOD Workshop on Networking Meets Databases, 2011.

[15] Simon Loesing, Martin Hentschel, Tim Kraska, Donald Kossmann: Stormy: An Elastic and Highly Available Streaming Service in the Cloud. EDBT/ICDT Workshops 2012: 55-60

[16] Nathan Marz: (Storm) Tutorial. https://github.com/nathanmarz/storm/wiki/Tutorial

[17] S4 Distributed Stream Computing Platform. http://incubator.apache.org/s4/

[18] Spark Streaming. https://spark.apache.org/streaming/

[19] Sankar Subramanian, Srikanth Bellamkonda, Hua-Gang Li, Vince Liang, Lei Sheng, Wayne Smith, James Terry, Tsae-Feng Yu, Andrew Witkowski: Continuous Queries in Oracle. VLDB 2007: 1173-1184

[20] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthikeyan Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy V. Ryaboy: Storm@Twitter. *SIGMOD 2014*: 147-156

[21] Trident: https://github.com/nathanmarz/storm/wiki

[22] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, Eric Baldeschwieler: Apache Hadoop YARN: Yet Another Resource Negotiator. SoCC 2013: 5

[23] ZeroMQ: http://zeromq.org. Retrieved December 1, 2014.