Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Bachelor's Thesis Nr. 226b

Systems Group, Department of Computer Science, ETH Zurich

Snailtrail for latency-sensitive workloads

by

Fabian Fischer

Supervised by

Desislava Dimitrova, Moritz Hoffmann, Timothy Roscoe

March 2018–September 2018

Contents

1 Introduction

The increasing amount of available data and the wish to analyze and react to this data in real-time led to a wide adoption of stream processing systems in modern datacenters. Stream processing systems target the processing of unbound streams of data. These streams are continuous flows of data items or events originating from various sources ranging from real world sensors over network metrics to application events such as new tweets on Twitter. Stream processing systems can then be used to react to certain events, aggregate the data or to perform arbitrary operations on the stream.

We can roughly differentiate between two key metrics in the performance of stream processors: latency and throughput. In stream processing system, we understand throughput as the number of processed data items in a given time frame. Latency is the time difference from receiving a data item until it is processed.

Especially for latency-sensitive applications, such as electronic trading, fast and reliable communication is crucial. In some cases latency increases of only a few milliseconds can be problematic. Much research [1, 2] focuses on reducing communication latency by exposing application-level information to the network which enables the network to better optimize for the problem at hand. We however argue that the network is often not a major source of latency and a detailed analysis of the given system often reveal more significant problems.

We take a deeper look into the latency characteristics of Heron, a real-time distributed stream data processing system developed at Twitter. After familiarizing with Heron, we quickly noticed unexpected processing latencies. A very simple application with a low load stream of one data item per millisecond showed a mean communication latency of up to 80 milliseconds, see Section 5.1, which is at least an order of magnitude more than expected.

We performed a detailed latency analysis of Heron for light workloads. For this we extended Heron with custom instrumentation that allows us to see the latency of the individual components of Heron. We found that configuring Heron is non trivial and that Heron's default configuration can cause a very high latency for some workloads. Further we found that in specific situations inaccuracies in the underlying libevent [3] library can cause a significant increase in latency. During the implementation of our instrumentation we also discovered that Heron does not stop some of its components gracefully and proposed a fix.

We will first look at related stream processing systems. This gives us an overview of different architectures, communication models and latency characteristics and helps us to put the results into perspective. We will then take a more detailed look at Heron and its implementation, which is necessary to understand the added instrumentation. Finally we will use this instrumentation to analyze Heron's latency characteristics.

2 Related Work

In this section we will look into different state-of-the-art *distributed stream processing systems*. We established in the introduction that stream processing systems consume unbounded streams of data and perform arbitrary operations on the data items in the stream. Distributed stream processing systems do exactly the same but in parallel, distributed over some kind of cluster.

The counterpart to distributed stream processing is *distributed batch processing*. The main difference between batch and stream processing systems is, that batch processing works on a bounded set of data and not on a stream of data. Otherwise these two share a lot of similarities and encounter similar problems. That is why there are systems that implement stream processing on batch processing systems, such as Apache Spark [4], and systems that enable batch processing on stream processing systems, such as Apache Flink [5].

For most distributed stream processing systems the behavior of an application is defined by a *logical plan* that can be compared to a logical query plan in a database system, in that it defines on a high level how a stream is processed. This logical plan is then converted into a *physical plan*, which defines how the actual computation is performed and how data is exchanged.

Physical plans and how the computation is performed, vary widely between different stream processing systems. For the logical plan however, most system use a *Directed Acyclic Graph (DAG)*. In this context a DAG is a collection of vertices, that represent computation, and directed edges, that represent data flow. As the name suggest, DAGs are not allowed to have circles. One notable exception is Timely dataflow [6], which allows its logical plan to have cycles.

2.1 Apache Flink

Apache Flink [5] both supports stream and batch data processing. It supports a lot of different APIs, all of these however compile down to a DAG called *dataflow graph*.

A Flink cluster consist of two different elements: The *JobMaster* and multiple *TaskManagers*. The JobMaster receives the dataflow graph from the client and schedules the operators on different TaskManagers. It keeps track of the state and progress of every operator and stream. In case of a node failure, it will use the tracked state to recover an earlier checkpoint. The TaskManager acts as a worker node and provides a number of tasks slot for operators to be scheduled on.

For data exchange TaskManagers have a direct communication with each other. Whenever a data record is created by an operator it is put into a buffer. The buffer is then sent to a consumer, as soon as either the buffer has reached its maximum size or when a timeout is reached. This allows us to configure Flink to either achieve low latency or high throughput. This is a fairly common design and we can see the same design in Heron [7].

Flink's latency is considered to be relatively low. While operating at maximum throughput, Flink can achieve a median latency in the order of ten milliseconds. [8]

2.2 Apache Spark Streaming

Apache Spark started as a research project [4] and was later donated to the Apache Software Foundation. Spark is in itself not a stream processing system but rather a general-purpose cluster computing system, with many high-level tools including GraphX [9] for graph processing, MLlib [10] for machine learning and Spark Streaming [11], which enables Spark to function as a stream processing system.

In Spark, users can manipulate so-called *Resilient Distributed Datasets* [4]. *RDDs* are collections of Python or Java objects. These collections are partitioned over the cluster and are resilient to node failure. On these RDDs users can execute operations like `map`, `reduce` or `filter`.

The key idea behind Spark Streaming is to split streaming computation into multiple batch computations on short time intervals. The data received during one time interval is stored in a RDD. The user can then execute multiple stateful or stateless operations on these RDDs, while treating multiple RDDs as a stream.

As Spark works on distributed datasets there is not the same notion of communication as in Apache Flink. Spark achieves high throughput, however latency is fairly high, usually in the order of several seconds [11].

2.3 Apache Storm

Storm [12] is a real-time fault-tolerant and distributed stream data processing system, initially created by Nathan Marz at BackType and later acquired by Twitter.

The basic Storm data processing architecture consists of streams of data items flowing through *topologies*. A storm topology is a directed graph where the vertices represent computations and the edges represent data flow between the components. Vertices are either spouts or bolts. Spouts are sources of input data. They typically pull data from queues and generate a stream of data items, called tuples. Bolts consume this stream of tuples and perform actual computations on the data. The processed stream is then potentially passed on to a set of downstream bolts.

Storm runs on a distributed cluster. Clients can submit topologies to a *master node* that is responsible for distributing and coordinating the execution of the topology. It schedules this topology onto *worker nodes*. These worker nodes perform the actual computation. Worker nodes run one or more instances of a JVM called *worker process*. Each of these worker processes is mapped to one topology. In each worker process run one ore more *executors*, which in turn perform one or more *tasks*. In these tasks, the actual work for a spout or a bolt is done.

Each worker has two additional threads dedicated to routing messages between workers. A *worker receive thread* and a *worker send thread*. The worker receive thread listens on a TCP/IP port and de-multiplexes incoming packets and queues the tuples to the incoming queue of responsible executor.

Each executor has two threads. One user logic thread and a executor send thread. The user logic thread receives the enqueued tuples from the worker receive thread, runs the actual task and forwards the processed tuple to the send thread. If the tuple is destined for a task on the same worker, it is directly sent to the correct executor. Otherwise the tuple is sent to the worker send thread, which then forwards it to the correct worker downstream.

Storms latency is considered to be low. Results vary depending on the benchmark but we can usually expect latencies of a few milliseconds which is even faster then Flink [8, 13]. Throughput however seems to generally be lower than that of Spark or Flink [8, 13].

2.4 Apache Heron

Heron [7] is a stream data processing engine developed by Twitter, as a direct replacement for Apache Storm. It tries to improve on it, while still being backwards compatible. We take more in depth look into Heron in Section 3.

As one key design goal was to maintain compatibility with the Storm API, Heron's data model and API is identical to that of Storm. So there are still tuples that flow through Heron topologies that are identical to Storm topologies.

While the data model remained identical to Storm's, the architecture was changed significantly. The main goals were to improve resource allocation, improve scalability and to simplify managing and debugging applications.

Heron splits the task of one of Storms worker process into a *Stream Manager* and multiple *Heron Instances*. A single Heron Instance only runs a single spout or bolt task and it runs in its own JVM. The Stream Manager on the other hand handles all communication. Instances only communicate with its local Stream Manager and do not have to handle routing of the data tuples they produce. The Stream Manager looks at all tuples it receives and sends them to

the correct next Stream Manager. This reduces the number of connections and should improve scalability of the overlay network.

Heron claims to have achieved higher throughput and even lower latency then Storm [7].

2.5 Dhalion

Dhalion [14] is a system that provides self-regulation capabilities to underlying streaming systems. It was implemented on top of Twitter Heron. There are other systems that provide similar capabilities, such as *DS2* [15] that enables auto scaling capabilities on Apache Flink [5] and Timely Dataflow [6].

Dhalion consists of three major components, the *Health Manager*, the *Action Log* and the *Action Blacklist*.

The Health Manager is a process that periodically invokes a policy, identifies potential problems and tries to resolve them. Dhalion differentiates between two different groups of policies: Invasive and non-invasive policies. Invasive policies actually react and change the topology (e.g. parallelism change). Non-invasive policies do not change the topology, but usually alert the user.

The Health Manager can execute multiple non-invasive policies but only one invasive policy at a time, as multiple invasive policies could result in conflicting actions.

The Action Log stores the actions taken by the Health Manager. This can be used for debugging and tuning a particular policy.

The Action Blacklist stores a set of actions and corresponding diagnosis that did not produce the expected outcome. These actions will not be repeated for a similar diagnosis. After an action was taken by a policy, the Health Manager evaluates the action that was taken. The system tracks the ratio of the number of times a particular action has not been beneficial. When the ratio is higher than a configured threshold, the diagnosis-action pair is placed in the Action Blacklist.

Before executing a resolver, the Health Manager checks if the action is already in the blacklist and does not invoke it if it is.

2.6 Photon

Photon [16] is not a general purpose stream processing system. It is designed to merge two log streams, similarly to a join operation in a RDBMS. At Google this system is used within the Google Advertising System to join streams of events such as web search queries and user clicks on advertisements. Photon provides low latency and datacenter-level fault-tolerance.

We look at the specific problem of joining two streams, web search queries and user clicks on advertisements. The system can also be used to join other streams with similar properties.

The same Photon pipeline is deployed in different datacenters. Each pipeline consists of three major components. The *Dispatcher* that reads user clicks on advertisements form a click log and then feeds them to the *Joiner*. The Joiner then finds the corresponding search query form the *EventStore*, a store that provides efficient lookup for these queries. Duplicates are avoided using a distributed key-value store called *IdRegister*.

Each piece of the topology communicates using RPCs. This provides lower latency than using a shared disk to communicate.

The end-to-end 90th-percentile latency of Photon is under 7 seconds [16]. We need to keep in mind that the round-trip time between their datacenters is up to 100ms. But throughput and fault tolerance seemed to have been more important than strong real-time guarantees.

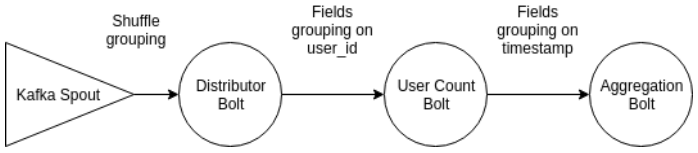


Figure 1: RTAC Topology

3 Heron

Twitter heavily uses real-time streaming for example to compute the real-time active user counts (RTAC). For a long time they relied on Apache Storm as their stream processing engine. Using Storm, Twitter encountered scalability, debug-ability, and resource sharing problems.

To tackle these problems, Twitter built a new real-time stream data processing system called Heron. Heron was developed as a direct replacement for Storm. It tries to improve some of the shortcomings of it, while still being backwards compatible.

In this section we will take a deep look into Heron's internals. We will concentrate on the parts which are relevant for the rest of this thesis. Most of the details come directly from Heron's source code [17], which is accessible on Github.

3.1 Data Model and API

One of the key design goals of Heron was to maintain compatibility with Storm's API. That means the data model and API of Heron and Storm are identical. This allows us to execute applications which were originally built for Storm.

The basic Heron data processing architecture consists of streams of *tuples* flowing through *topologies*. Tuples are arbitrary data items. A tuple usually represents something like a single sensor measurement or an application event, like a click on a advertisement or a tweet on Twitter. A Heron topology is a directed graph where vertices either represent sources of data or computation of the stream, and the edges represent the data flow.

We call the vertices that produce data *spouts*. They typically pull data from queues, such as for example Kafka [18], and generate a stream of tuples. Vertices that consume this stream of tuples and perform actual computations on the data are called *bolts*. The processed stream is then potentially passed on to a set of downstream bolts. Both bolts and spout are arbitrary Java code.

Figure 1 shows an example topology that counts the number of active users in real-time. It gets the necessary data from Kafka, distributes the data to multiple nodes, groups the tuple by user id and timestamp, and can then counts the active users in a distributed manner.

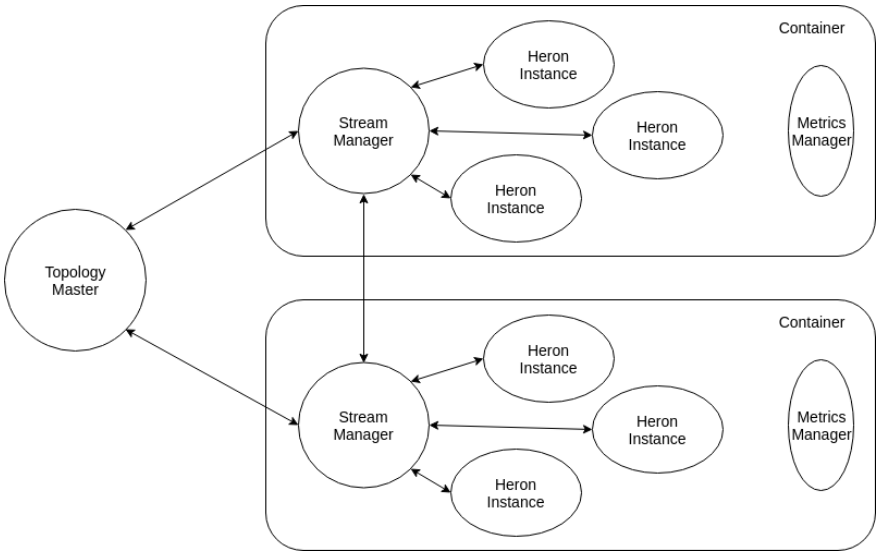


Figure 2: Heron Topology Architecture

3.2 Architecture

While the data model remained identical to Apache Storm's, the architecture has been changed significantly.

Unlike Storm, Heron can run on different general purpose distributed systems, and does not rely on a specialized cluster scheduler like Storm's Nimbus. Twitter runs Heron on Aurora [19] using their homegrown scheduler.

That means each topology is run as an Aurora job consisting of multiple *containers*, see Figure 2. Containers are an abstraction of a separate machine. Multiple containers can run on a single physical node. The first container runs the *Topology Master*. The other containers run a *Stream Manager*, a *Metrics Manager*, and multiple *Heron Instances*.

The Topology Master manages the topology throughout its runtime and provides information on the status of the topology. It also acts as a gateway for metrics, but is not involved in any actual data processing.

Heron Instances does the main work for a bolt or a spout. That means it either produces or consumes tuples. All data movement and routing complexity was moved to the Stream Manager. Unlike a Storm worker, a Heron Instance runs in a dedicated JVM.

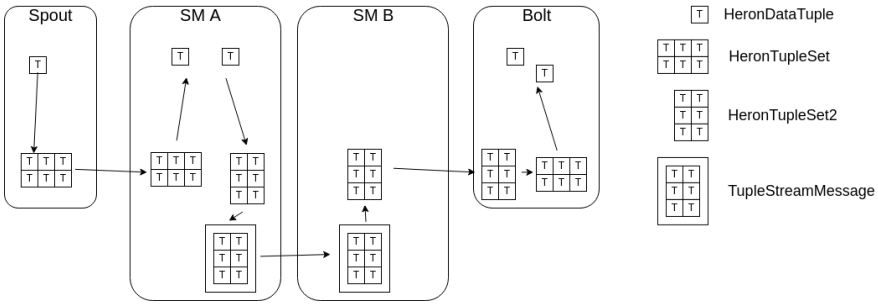


Figure 3: Overview of the communication model

The Stream Manager is responsible to efficiently route tuples. Each Heron Instance connects to its local Stream Manager to send and receive tuples. Each Stream Manager connects to all other Stream Manager in the Topology and forms a $O(k^2)$ connection network, where k is the number of containers. The number of Heron Instances n can be a lot larger than k. This allows the overlay network to scale more efficiently.

Unlike Storm, Heron implements a *backpressure mechanism*. This means the rate at which data flows through the topology is dynamically adjusted if the downstream stages are running slow. This is done to prevent tuple loss and unused work. When the Stream Manager realizes that one or more of his Heron Instances is slowing down it stops reading data from its local spouts and sends a special start backpressure message to all other Stream Managers requesting them to do the same. Once the Heron Instance caught up it resumes reading from its spouts and sends a stop backpressure message to all Stream Managers. Heron calls this the *Spout Backpressure* approach.

As the Stream Managers performance is critical, it was written in C++. Underlying it uses the libevent [3] eventlooper to react to incoming tuples, enqueue them in the correct buffer and forward them in batches.

The Metrics Manager collects and exports any metrics collected on the topology. There is one Metrics Manager for every container. The collected metrics are then passed to a central monitoring system and to the Topology Master.

3.3 Communication Model

In this section we will focus on the interaction between the different components of Heron. We will take a more detailed look into these components later on.

Heron's components use language neutral objects called *protocol buffers* [20] to communicate between each other. You can think of protocol buffers to be similar to C structs, in that they contain structured data with one or more fields. These protocol buffers can be mapped to objects for most major programming

languages. This allows a Heron Instance, written in Java, to easily communicate with the Stream Manager, which is written in C++.

Figure 3 represents a spout and a bolt on two different containers that communicate with each other. We established that spouts produce data tuples. These are represented as a protocol buffer called `HeronDataTuple`. However nearly immediately after producing, the tuples are grouped into tuple sets. These tuple sets are themselves also represented as a protocol buffer called `HeronTupleSet`. They are then sent to its local Stream Manager via a TCP socket.

When the set arrives at the first Stream Manager, it is broken down into its tuples. The Stream Manager then performs the necessary routing decisions and builds a new tuple set for every destination. This time the tuple set is a different protocol buffer called `HeronTupleSet2`. There are only minor differences between these two kinds tuple sets. Before being sent out the tuple set is then wrapped into another protocol buffer called `TupleStreamMessage` which is then sent to the correct next Stream Manager.

The second Stream Manager then extracts the tuple set from the received stream massage and forwards the tuple set to the correct bolt. When the tuple set arrives at the bolt, it will actually convert the `HeronTupleSet2` into the `HeronTupleSet` protocol buffer and only then it will read the tuples from the tuple set and process them as individual tuples.

The communication between two Instances is not as straight forward as it initially seems. The tuples are wrapped into tuple sets or other messages for most of their journey. This makes some of the questions, like "How long does it take for a tuple to reach the second Stream Manager?" or "What is the network latency per tuple between the Stream Managers?", either computationally too expensive to answer or simply nonsensical.

3.4 Mertics in Heron

Heron already provides quite a few different metrics that can either be exported to multiple monitoring systems such as Prometheus [21] or some of them can be viewed in the *Heron UI*, a web interface shipped with Heron that provides a quick overview of the topology.

As we focus on the latency characteristic of Heron, we should look what kind of relevant metrics Heron already provides.

There is the *completeLatency* which is the moving average latency of a tuple, from being produced at the spout, until it is acknowledged. This includes the production of the tuple at the Spout, the computation at the Bolts, all the communication latency of the tuple and the time the acknowledgement tuple takes to return to the Spout. This can be helpful to get an impression of the latency or to notice an error with the application, but a single moving average

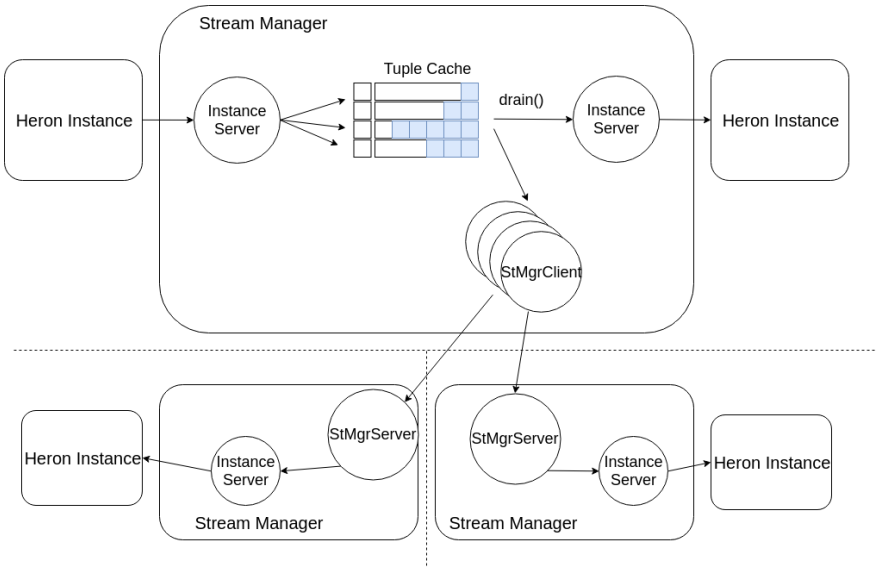


Figure 4: Overview of a Stream Manager

does not provide sufficient insight into Heron's latency. We can neither see the tail latency nor notice where the latency is coming from.

Heron also exposes the *executeLatency*, *processLatency* and *failLatency*, which are the mean latencies to process a single tuple, acknowledgement and nack respectively. These metrics give the same limited insight as it only shows a single mean latency and does not tell us anything about the tail latency. Also these metrics only measure the execution time of the user code and not of Heron itself or the network latency. This again makes these latencies useful for debugging the topology, but not for getting insight into Heron itself.

The rest of the metrics do not give us relevant latency information, but are mostly counters of sent, processed or received tuples or information on buffer sizes.

3.5 Stream Manager

We will take a deep look into the internals of the Stream Manager. Figure 4 shows an overview of a Stream Manager and its communication with other Stream Managers. We will look into three main components: The `Instance Server`, which communicates with the local Instances, the `TupleCache`, which acts as a buffer, and the `StMgrClient` and `StMgrServer`, which handle the connection between the Stream Managers.

The Stream Manager is built on the libevent [3] eventlooper. It executes a callback function when a message arrives or after a timeout has been reached.

3.5.1 InstanceServer

When either a spout or a bolt emits a tuple it put into a collection of tuples, called tuple set, which acts as a buffer. The Heron Instance does not differentiate between tuples with different destinations. This means in a tuple set each tuple may have a different destination. The tuple set is then sent to the local Stream Manager.

Whenever a message arrives at the Stream Manager the Eventlooper will then call the correct handler. When tuple set is received the `InstanceServer` will iterate over every tuple in the tuple set and make the routing decision for each of them. It then uses this routing decision to add the tuple to the corresponding `TupleCache`.

3.5.2 TupleCache

The `TupleCache` is basically a dictionary that maps the `task_id` of the destination to a `TupleList`. That means at every Stream Manager there is a `TupleList` for every Heron Instance. Each of these `TupleList`s consist of a queue of tuple sets and a `current` tuple set. When a tuple is added to a `TupleList`, it is first added to the intermediate `current` tuple set until it exceeds a configured size, then the complete tuple set is added to the queue. If the total maximal size of the cache is exceeded, we call the drain function on every tuple cache and send out all the tuple sets.

To prevent a buildup of tuples in low load situations, the event loop is scheduled to regularly drain the `TupleCache`, which means it will send out all the tuple sets in the queue. This does not include the `current` tuple set. It is only drained, if the queue is empty and it did not send out anything the last time. So in a low load situation, the `current` tuple set will be drained every other time.

If the destination instance is on the same container as the Stream Manager it will be sent to it directly using the `InstanceServer`. Otherwise it will be sent to the correct Stream Manager using the `StMgrClient`. This seems to be the *short-circuiting mechanism* mentioned in the paper [7].

3.5.3 StMgrClient & StMgrServer

The `StMgrClient` and `StMgrServer` handle the communication between the Stream Mangers.

On start of the Stream Manager, the Stream Manager creates an instance of `StMgrClient` for all other Stream Manager and opens a TCP socket for each of them.

This means every Stream Manager has a socket connecting it to all other Stream Managers. Now when a Stream Manager wants to send a tuple set to the next Stream Manager, it will call the correct client and send the message to the corresponding Stream Manager.

The receiving Stream Manager will then forward the incoming tuple set to the correct Heron Instance through the `InstanceServer`.

3.6 Heron Instance

In this section we will take a closer look at Heron Instance. Figure 5 is an overview of a Heron Instance running a spout task. A Heron Instance running a bolt task would look very similar.

A Heron Instance runs in two threads, a *Gateway thread* and a *Task Execution Thread (TET)*. The Gateway handles all communication with the Stream Manager or Metrics Manager. The Task Execution Thread either runs as a spout or bolt.

The two threads are connected by three queues, the `data-in`, `data-out` and the `metrics-out` queue. Data-in and data-out queues are bound in size. If the data-in queue exceeds this bound, the backpressure mechanism is triggered.

3.6.1 Task Execution Thread

Both the TET and the Gateway thread use an event loop implementation similar to *libevent* used by the Stream Manager. Unlike *libevent* this event loop implementation was specifically written for Heron. The `WakeableLooper` executes in a while loop until it is explicitly stopped. The loop allows to register timer events and tasks on wake up. Both of them are arbitrary `Runnables`. At every iteration of the loop the wake up tasks and all expired timer events are executed.

The TET uses the `SlaveLooper` which is an extended `WakeableLooper`. A spout and a bolt only really differ in what tasks this looper is executing.

Spout For users to able to run their own spout code, they need to implement the `ISpout` interface. This mainly means, it needs to provide a `nextTuple()` method which produces a single tuple and calls `emit()` on a provided output collector. With that the produced tuple is added to the `OutgoingTupleCollector`, which acts as a buffer that collects the tuples and groups them to a tuple set. If the size of the tuple set exceeds the configured tuple set size, the collector will enqueue it into the `data-out` queue.

On startup the spout will register multiple tasks, which in turn will run on every iteration of the event loop. One of the tasks is to call the user provided `nextTuple()` in a loop. This will fill up the tuple collector, as long as the configured size or timeout is not exceeded. If one of these is exceeded it will flush the current tuple set to the `data-out` queue. We have seen a very similar design in the Stream Manager, where a buffer is flushed when either a time or size barrier is exceeded. This design decision is used throughout Heron.

The spout also handles incoming control packets, for example acknowledgements for sent tuples. However this is not relevant for the thesis and will not be covered in more detail.

Bolt Similarly to how spout code is executed, for a user to be able to run his own bolt code, he needs to implement the `IBolt` interface. The most important function of this interface is the `execute()` function. This function takes a single

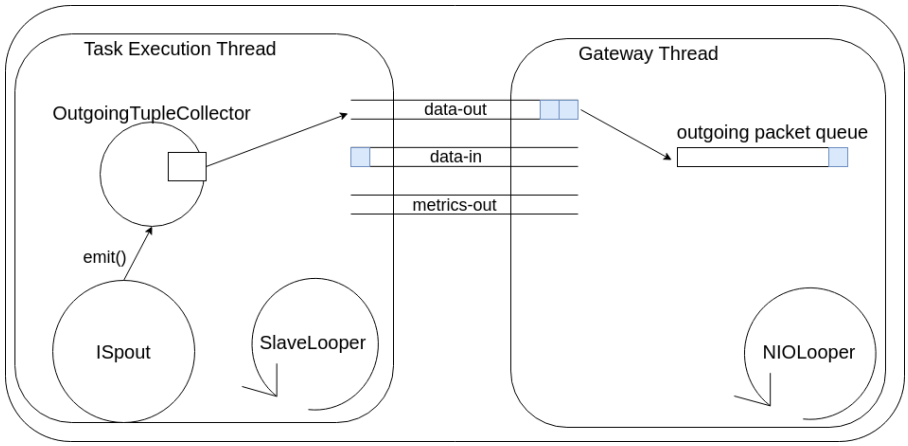


Figure 5: Overview of a Heron Instance

tuple and processes it. Similarly to the `nextTuple()` of the spout, it can then enqueue a tuple into the `OutgoingTupleCollector`.

The bolt also utilizes the `SlaveLooper` and registers tasks to run on wake up. It will read from the `data-in` queue and call `execute()` on every tuple until the queue is empty or a configured time is exceeded. After that it will flush the current tuple set.

3.6.2 Gateway Thread

The Gateway Thread builds on the `NIOLooper`, which is again just a slightly modified looper. The Gateway mainly does two things: Read from the `data-out` queue and send it to the Stream Manager, and read from the Stream Manager and enqueue it into the `data-in` queue. It does this with two tasks which are both scheduled to run at every iteration of the event loop, the *sending task* and the *receiving task*.

The sending task basically moves tuple sets from the `data-out` queue of the Heron instance to an outgoing packet queue, from where they are sent to the Stream Manager. If there are no outstanding packets in this queue, it will copy tuple sets from the `data-out` queue to the outgoing packet queue until the `data-out` queue is empty. This basically means it copies the whole `data-out` queue into the outgoing packet queue.

This will also add another task to the looper, that sends out a batch from the outgoing packet queue. As soon as the queue is empty, the task is removed from the looper.

The receiving task reads packets from the TCP socket and enqueues them into the in-queue. It does this similarly to sending task in that it schedules a

`read()` task, whenever the in-queue is ready to receive new tuples. This `read()` task does the actual reading.

So when reading is enabled, the looper reads a batch of incoming packets, parses them and handles each packet accordingly. That means in case of a tuple set, it enqueues it into the in-queue.

4 Instrumenting Heron

Our goal is to find out which parts of Heron are responsible for the observed latency. In this section, after a detailed look into Heron's architecture, we try to identify possible latency sources. We look whether the metrics provided by Heron can already give us a good idea, which components have a major impact on latency. Finally we present our additional instrumentation and why we chose to implement them.

In Figure 6 we can see a representation of simple tuple flow between a single spout and bolt, located on two different containers, including every major buffer. A tuple flow between a spout and a bolt located on the same container looks very similar. It simply does not pass through the second Stream Manager.

As seen in Section 3.4, Heron already provides some metrics on the latency. However neither do they provide any insight on the source of the observed latency, nor do they show the latency distribution, which would be necessary to properly analyze Heron's latency characteristics.

4.1 End-to-End Latency

In order to notice any problems with Heron's latency, we will need to collect the end-to-end latency for each individual tuple, from spout to bolt.

To get this end-to-end latency, we do not need to alter Heron, we did this by writing a simple custom topology consisting of a single spout and bolt. The spout takes the current timestamp in nanoseconds and emits it. The bolt receives the timestamp, calculates the latency, and stores it. This works well as long as the spout and bolt are on the same machine, otherwise precise time synchronization between the hosts is needed.

To reduce the amount of data collected, we decided to use a high dynamic range histogram [22], or HDRHistogram. It allows us to record a histogram of the latency without any noticeable performance impact. The resulting histogram is written to a file as soon we processed a configurable amount of tuples

4.2 Network Latency

To notice whether latency issues originate from the network itself, we want the latency distribution of the messages between the Stream Manager and the Heron Instances and between the Stream Managers themselves. We need to keep in mind that these messages contain a tuple set and not individual tuples. Doing this will require us to extend the protocol buffers to add timestamps to the messages.

For the instance to Stream Manager communication, we first extend the `HeronDataTupleSet` protocol buffer by adding an optional `timestamp` field. With this, we can change the `sendMessage()` function in the `HeronClient` class to take a timestamp before sending the message out at the instance. This

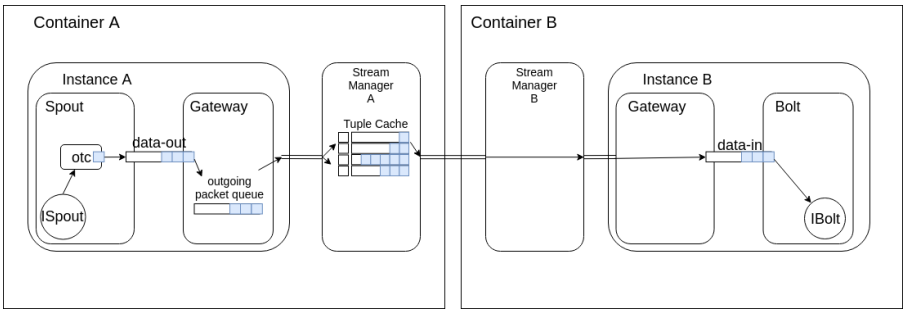


Figure 6: Dataflow between a spout and a bolt, located on two different containers

will add a timestamp to every tuple set added to the outgoing packets queue from where it will be sent to the Stream Manager. This means there is still a buffer between taking the timestamp and actually sending the message, however this buffer is tightly integrated and works on `ByteBuffer`, which makes adding the timestamp after the buffer impractical as we would have to first parse the `ByteBuffer` before re-encoding it for every tuple set, which we expect to generate too much overhead. We will need to keep this in mind for the evaluation.

We then calculate the latency directly when receiving the tuple set at the instance server of the Stream Manager. The instance server already needs to read every tuple in the set, so we do not expect to notice any performance impact at the Stream Manager. The latency is then stored in a HDRHistogram and printed when the Stream Manager exits.

For the Stream Manager to Stream Manager communication another protocol buffer is used, the `TupleStreamMessage` so we again need to extend this buffer with an optional timestamp field. Now we add a timestamp to the message right before sending it from the `StMgrClient`. This is also where the `TupleStreamMessage` is built, so there should not be any performance overhead from this. At the receiving Stream Manager, we calculate the latency and build a HDRHistogram before sending the tuple set inbound. With this setup we not only collect latency information on data tuple sets but also on control tuple sets as they are treated the same way in the Stream Manager to Stream Manager communication.

The constructed HDRHistograms are written to files, as soon as the topology is stopped. However the Stream Manager is not stopped gracefully in Heron, which prevents us from doing that. We consider this a bug in Heron and fixed this for this thesis, as we needed it to collect most of our metrics. We created a patch to enable graceful shutdown of the Stream Manager and it is waiting to be pushed upstream [23].

We opted not to record the Stream Manager to Instance communication, as we expect similar results as in the Instance to Stream Manager communication.

4.3 Instance to Stream Manager

We collected the end-to-end latency for every tuple. To be able to pinpoint where each tuple spends how much time, we want to split this complete latency into sections, from the emission of the tuple until different points in the tuple flow. This would give us easily comparable results.

The fact that Heron batches multiple tuples in tuple sets however makes this difficult. Right after emitting a tuple at a bolt or spout the tuple is packed into a set. This tuple set is then sent to the Stream Manager, where the tuples are rearranged into completely different tuple sets and sent to the next Instance. This makes tracing a single tuple throughout the data flow nearly impossible, without introducing large performance penalties.

There is however one point we can easily instrument on a per tuple level, without introducing a lot of overhead. When a tuple set arrives at the first Stream Manager, it is split into tuples, which are then routed to the correct destination. At this point we can easily get the timestamp for each tuple.

To do that we extended the `HeronDataTuple` protocol buffer with an optional timestamp field. Now right after the spout or bolt emits a tuple and before adding it to the `OutgoingTupleCollector`, the actual protocol buffer for the tuple is built. There we can easily add a timestamp without adding any overhead At the Stream Manager, before it makes the routing decision for every tuple, we calculate the latency and constructed a HDRHistogram, which is written to file when stopping the topology.

Contrary to our other measurements, which are on a tuple set granularity, this gives us results which are directly comparable to the end-to-end latency. This way we can easily see whether the observed end-to-end latency originates at the spout or not.

4.4 Tuple Cache

We collect metrics for the actual network communication, but what is also interesting is how long a tuple set spends in a buffer before being sent. One of such a buffer we expect to have an impact on latency is the tuple cache. At each Stream Manager there is a tuple cache with queues for every Heron Instance. When the routing decision is made, the tuple is enqueued into the correct queue. The tuple cache builds tuple sets on the fly and buffers them until they are sent out. There is a more detailed write up on the tuple cache in section 3.5.2

We want to measure the time from enqueueing a tuple into the tuple cache until it is sent out. This is relatively difficult, as the tuples are added individually and sent out as a tuple set. We can however collect two other metrics without introducing a lot of overhead: The time it takes to build a tuple set and how long a tuple set stays in the cache. With this data we should be able to get a good impression on how long each tuple remains in the tuple cache.

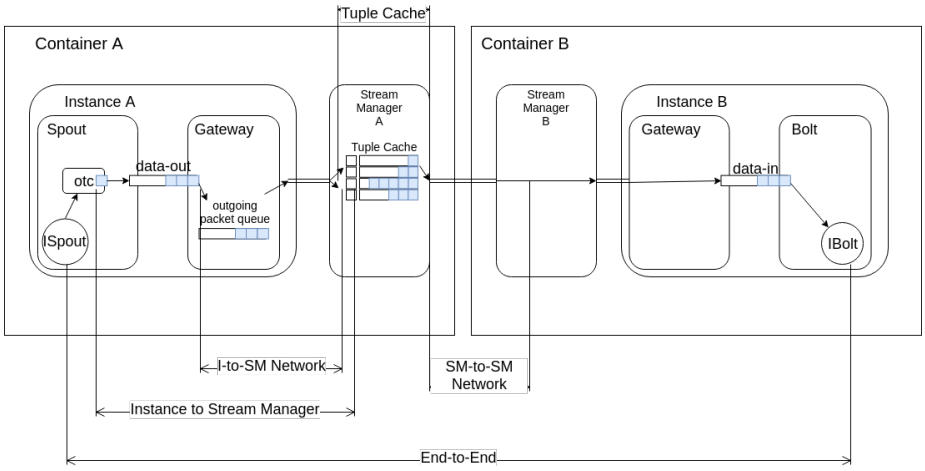


Figure 7: The instrumentation added to Heron

We get these metrics the following way. We add a timestamp to a tuple set as soon as we add the first tuple to the `current` tuple set. We do this by extending the `HeronDataTupleSet2` protocol buffer with a timestamp field. We then calculate the build latency as soon as the `current` tuple set is either added to the queue or flushed by the `drain()` function. When any tuple set is flushed we use the already added timestamp to calculate how long the tuple stayed in the buffer. Both latencies are again added to HDRHistograms and written to a file as soon as we terminate the topology.

This effectively measures the time the first tuple added remains in the `current` tuple and how long it takes until it is flushed from the cache. We need to keep in mind that the mean time any tuple spends in this cache will be lower. But by subtracting half of the build time we should be able to estimate this mean time.

4.5 Gateway Thread

Another buffer that might be responsible for an increase in latency is the queue between the task execution thread and the gateway thread. The task execution thread collects the produced tuples in the `OutgoingTupleCollector` and enqueues them as a tuple set into the `data-out` queue. The gateway thread then reads from this queue and sends out the tuple sets in batches.

This is in some way very similar to the tuple cache in the Stream Manager. It would be interesting to see how long it takes to build a tuple set and how long the tuple sets rest in the `data-out` queue. We did however not implement this instrumentation, as we could already get an impression of this latency from the Instance to Stream Manager latency and we soon noticed that this buffer usually does not seem to be a major source of latency.

5 Evaluation

To examine Heron's latency behavior, especially in low load situations, we performed multiple micro benchmarks.

All benchmarks run on a local cluster on a node with 4 AMD Opteron Processor 6174 and 128GB of main memory.

5.1 End-to-End Latency

As we have already described in the introduction section, we observed a very high end-to-end latency for very simple, low-load topologies. We try to reproduce this observation. Also, instead of relying on the moving average of the end-to-end latency reported by Heron, we build a histogram of the end-to-end latency.

Without prior knowledge we would actually expect a low end-to-end latency of only a few milliseconds, as there is no computation that should slow down the system. However in line with the previously observed latency we expect a high latency of around 80 milliseconds.

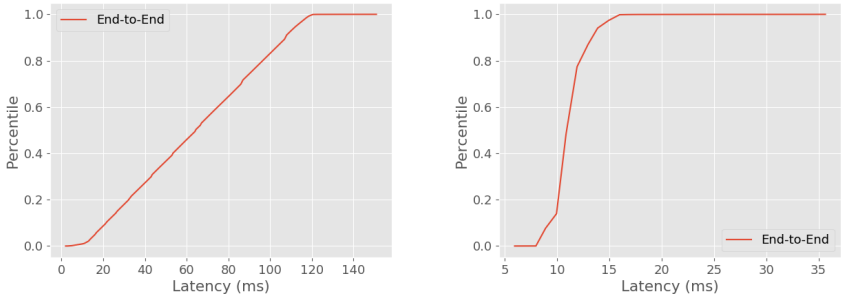
We are using a very simple topology as described in Section 4.1. The topology consists of a single spout and a single bolt. The spout waits for a millisecond and emits a configurable amount of tuples containing the current timestamp. The bolt receives the timestamp, calculates the latency and inserts it into an HDRHistogram.

We run the topology twice. The first time, we only emit a single tuple per millisecond. This is a very low load that should uncover any problems connected with this type of system usage, while still being something we can reasonably expect to see in a distributed stream processing system. As an example, Reddit, the growing social news and media aggregation website, saw about 8.89 billion *upvotes* in 2015 [24]. An upvote is an indication by a user that he liked a certain article. This means Reddit saw a mean of nearly 300 upvotes per second which translates to about one upvote every 3 milliseconds. This is comparable to the load we put on our topology and it would be entirely reasonable for Reddit to use a distributed stream processing system to analyze their data.

In the second run we emit 500 tuples every millisecond. We found that at this load most effects connected to very low system usage disappear without causing other problems connected to high loads.

The topology runs in two containers with 4 CPUs. The spout and bolt each run in a separate container and get 4 GB of RAM. With this configuration the RAM and CPU allocation should be more than enough to handle the topology and we should not see effects caused by underprovisioning. Both containers run on the same machine so there should not be any problems caused by clock synchronization inaccuracies.

We use a unmodified Heron version 0.17.5 with default configuration. We do this mainly to get a baseline of an unmodified system for reference.



(a) 1 tuple/ms

(b) 500 tuples/ms

Figure 8: CDF of the End-to-End latency of a very simple topology consisting of a single spout and bolt with running on unchanged Heron.

Figure 8 shows the latency histogram collected by the bolt for the two runs. For the first run with very low load, we can see the tuple latency is more or less uniformly distributed from 10 ms to about 120 ms. Heron's self reported average end-to-end latency is at a little less then 80 ms. For the second run with 500 times higher load we actually see a reduced end-to-end latency of around 10ms to 20ms. The mean complete latency reported by Heron is at about 22 ms.

The second run is more or less as expected. The mean complete latency reported by Heron is a bit surprising, although it does make sense when we include the fact that the complete latency of Heron includes the time the acknowledgment takes to return to the spout. So a complete latency of up to double that of our measurements is expected.

In the first run we were able to reproduce the previously observed effect. The latency is significantly higher then we would expect from such a simple, low load topology. There is also a huge variance in latency. We expect this to come from a design decision of Heron. In Heron each buffer is either flushed at a configured capacity or after a configurable time. Keeping this in mind, we can explain these two graphs with a misconfigured buffer. The second run with its increased load hits the capacity threshold and flushes the buffer in a reasonable time. The first run however does not fill the buffer and is flushed by the timeout which seems to be set too high. This could result in such a wide distribution, as the last tuple added to the buffer would have a low latency and the first a very high and everything in between would be uniformly distributed.

5.2 Added Instrumentation

To find the source of the observed latency, we are adding the instrumentation described in Section 4. We expect more or less the same results as with the unmodified Heron. There might be a minor increase in latency caused by the overhead of our instrumentation. There are multiple buffers that could be responsible, in the Stream Manager as well as in the Instance.

We repeat the first measurement again using the simple topology consisting of a single bolt and spout. The topology again runs in two containers with 4 CPUs and 4 GB RAM for both spout and bolt and we again run it two times. Once sending a tuple every millisecond and once sending 500 tuples every millisecond. We use a modified Heron version 0.17.8 with added instrumentation described in Section 4. We still keep the same default configuration.

Figure shows 9 the latency histograms collected by our instrumentation. For the low load run we can see the tuple latency in figure 9a. The end-to-end latency did not change significantly and ranges from 10ms to 120ms. The Instance to Stream Manager latency is a bit lower at about 1ms to 100 ms and equally distributed as the end-to-end latency. In Figure 9b we can see the tuple latency for the higher load situation. The Instance to Stream Manager latency is very low at about 3-4 ms. The end-to-end latency is at about 10ms to 20ms, so about 5ms higher than without the instrumentation

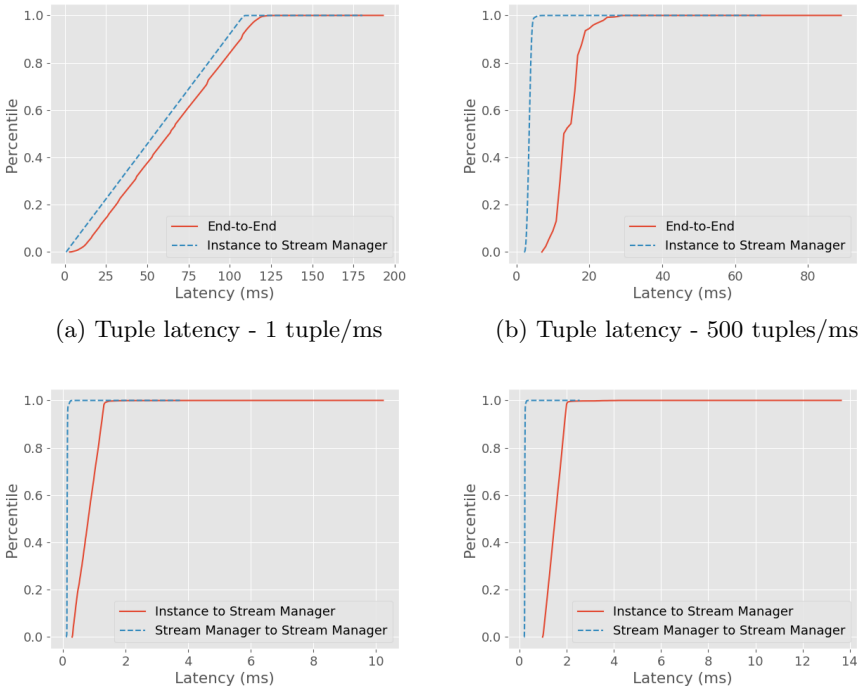
The actual network latency can bee seen in Figure 9c and 9d. The network latency is very low. For both runs the Stream Manager to Stream Manager latency is below 1 ms. The Instance to Stream Manager is at 1ms to 2ms.

From this we can deduce that most of the increased latency for the low load run seems to be coming from the out-queue between the task execution thread and the gateway thread in the spout. We loose up to 100ms or more at this step. As we guessed this looks very much like a misconfigured buffer. After a look into the *heron_internals.yaml* configuration file, which configures multiple parts of Heron including all timeouts for buffers, we can see that by default for local clusters the timeout for the out-queue is set to 160ms. This seems to create the latency we observe. The latency is still lower than 160ms so it looks like the buffer actually hits the capacity threshold after about 100ms and is flushed.

For other kinds of clusters this configuration is set to 16ms, so we actually assume this to be a bug in the default configuration. In general all other buffer timeouts are set to 16ms, which we still find too high for low latency workloads.

As for the higher load run, not much seems to have changed. There is a slight increase in end-to-end latency. As this increase cannot be seen in the Instance to Stream Manager latency, we assume this is cause by some overhead at the tuple cache instrumentation.

The network latency does not seem to cause a significant part of the observed latency. The higher latency of the Instance to Stream Manager communication is most likely caused by the `ByteBuffer` cache described in Section 4.2.



(a) Tuple latency - 1 tuple/ms

(b) Tuple latency - 500 tuples/ms

(c) Network latency - 1 tuple/ms

(d) Network latency - 500 tuples/ms

Figure 9: Latencies of a very simple topology consisting of a single spout and bolt with running on Heron with added custom instrumentation

5.3 Updated Configuration

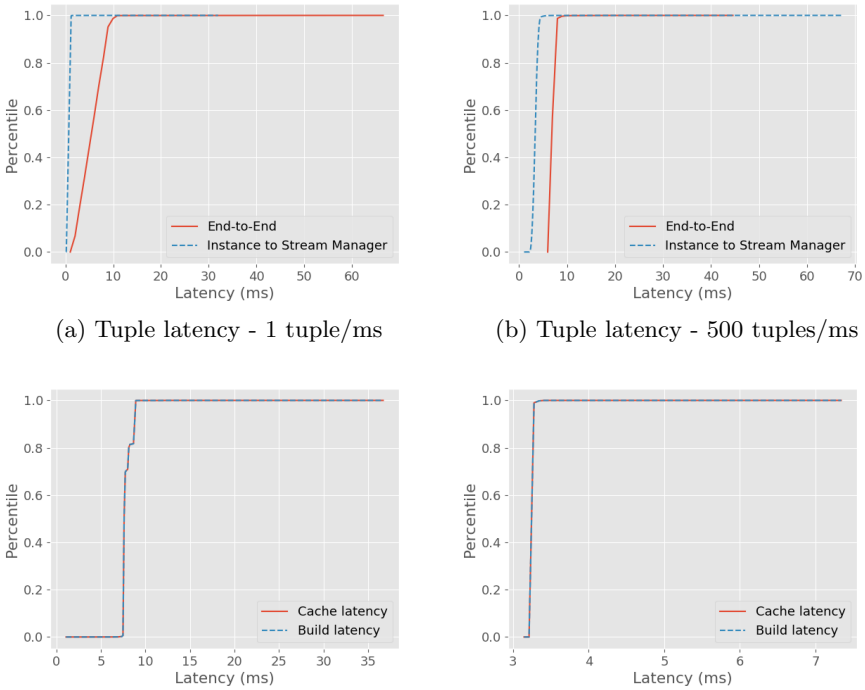
By updating the configuration and decreasing all timeouts we should be able to cause a large drop in latency for the low load run. We expect a consistently low end-to-end latency of a few milliseconds, lower then the higher load run in the last experiment. As for the higher load run we might actually see worse performance as we force buffers to flush early. We do however not expect to see a large difference.

We again repeat the same measurement using the simple topology consisting of a single bolt and spout. The topology again runs in two containers with 4 CPUs and 4 GB RAM for each spout and bolt and we again run it two times. Once sending a tuple every millisecond and once sending 500 tuples every millisecond. We use the same modified version of Heron. We did however update Herons configurations. To achieve lower latency for the low load run, we dropped all buffer timeouts to one millisecond. This especially includes the timeout for the data-out queue, which we found to be massively misconfigured by default.

Figure 10 shows the collected latency histograms. As expected for the low load run the latencies reduced drastically. The end-to-end latency is now between 2ms and 10ms. The Instance to Stream Manager latency dropped to about 1ms. For the higher load run, not much seems to have changed. The end-to-end latency actually seems to have dropped slightly to about 8ms.

For both runs the cache latency, that is the total time a tuple set remains in the tuple cache, is equal to the build latency, the time it takes to build the tuple set at the tuple cache. For the low load run every tuple set remains in the cache for about 8ms to 10ms. For the higher load run this latency is lower at a little over 3ms.

The end-to-end latency of the low load run dropped significantly by about a factor of ten. However tuple sets seem to wait in the tuple cache for up to 10 milliseconds, although the tuple cache is configured to be flushed every millisecond. We need to keep in mind, that the measured time a tuple set is in the cache is equal to the time the first tuple added to the set spends in the tuple cache. This explains that for the most tuples the end-to-end latency is actually lower than the mean time a tuple set is in the tuple cache, but we can clearly see the linear distribution from about 2 millisecond up to 10 milliseconds which we assume to be caused by this delayed flushing of the tuple cache. This would mean that most of the observed latency is caused by this single buffer. It is also interesting to see that for both runs the cache latency is equal to the build latency. This means the tuple sets are never actually inserted into the queue but are always flushed from the `current` tuple set, or that they are flushed nearly immediately after being enqueued into the queue.



(a) Tuple latency - 1 tuple/ms

(b) Tuple latency - 500 tuples/ms

(c) Cache latency - 1 tuple/ms

(d) Cache latency - 500 tuples/ms

Figure 10: Latencies of Heron with added instrumentation and reduced buffer timeouts

5.4 Libevent

We noticed that the tuple cache is only flushed every 10 milliseconds instead of the configured one millisecond. As we covered in Section 3.5.2 the tuple cache is flushed by a libevent timer that is now configured to run every millisecond. However the currently building tuple set is only flushed if we did not flush anything last time. So in case of very low load, we only flush the cache every second iteration. This means the cache would flush every 2 milliseconds, which still does not explain the observed 10 milliseconds.

It would be quite surprising, if libevents timer were the source of the observed latency, especially since libevent configures its timer in microsecond granularity. It would however explain our observation well.

To test whether the underlying libevent library is the root of this inaccuracy, we wrote a standalone piece of C++ code, that registers a libevent timer to run periodically. It then measures the actual period and constructs an HDRHistogram. We run it for timer periods between 1 microsecond up to 50 milliseconds. We run each measurement for 30 seconds.

Figure 11 shows the measured period with the actual configured period as as vertical line. For periods below 2 milliseconds, the actual measured latency lays much above (close to 5ms) the configured flush interval, indicating inconsistency in behaviour. For 2 ms up to 10 ms, the distribution is more or less correct, however there is still quite a high variance. For periods of 10 ms and above, the accuracy is starting to improve and is reliably close to the configured period.

Libevent does seem to be the source of the observed latency. Libevent timers are not suitable for periods below 5 milliseconds and always seem to run every 5 millisecond if configured for lower periods. This explains the 10 millisecond delay in Heron, as the current tuple set is only flushed every second time and the configured 1 millisecond timer only runs every 5 millisecond.

In conclusion we can see, that most of the observed latency was caused by an error in the standard configuration of Herons local cluster. This prevented a buffer to be flushed in time. Further tuples are stuck in the tuple cache for up to 10ms because of inaccuracies in the underlying libevent library. We were however not able to pinpoint the cause of this inaccuracy.

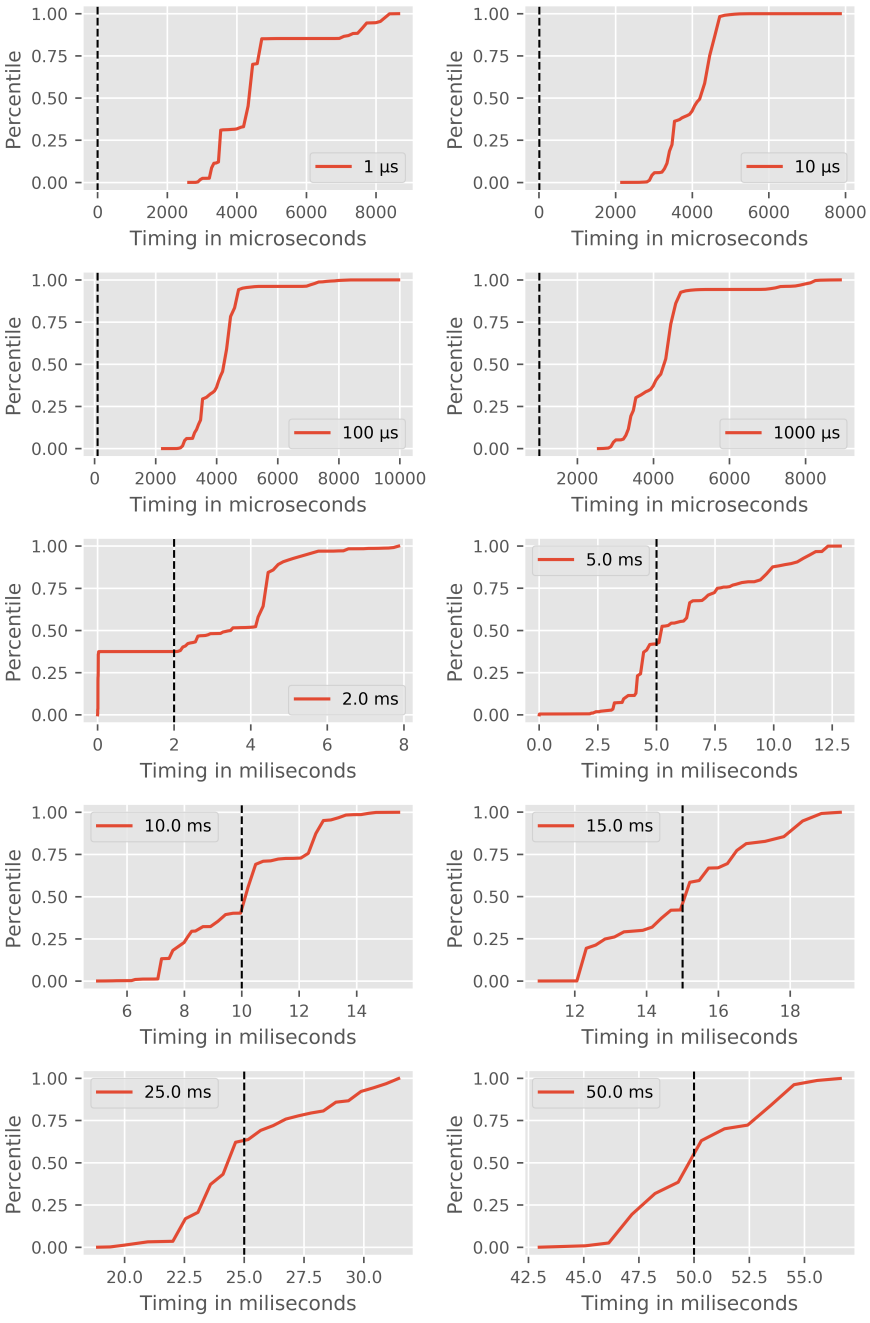


Figure 11: CDF of the Latency of libevent timers. The vertical line represent the actual configuration.

6 Conclusion and Future Work

Our goal was to explain the unexpectedly high observed latency. We added more detailed latency instrumentation and we found that as expected most latency is added by the various buffers inside Heron and not by the network. Heron's buffer architecture results in hard to predict latency characteristics and makes it difficult to configure.

One example of this complex and hard to predict system is that with Heron's default configuration we can actually sometimes observe that latency increases when we decrease load, caused by the misconfiguration of a buffer. And even after tuning Heron's configuration for low latency workloads we can see a similar effect caused by inaccuracy in the timing of the underlying libevent library.

We believe some design decisions, especially the liberal usage of buffers and the design of these buffers, made Heron hard to reason about and had a negative impact on latency. Nearly every stream processing system uses batching to increase network throughput, but Heron batches tuples into tuple sets, places these sets into buffers which are then sent in batches. It is not clear whether this double batching actually improves or hinders performance.

Reworking some parts of Heron could improve its latency characteristics. Multi-threading the Stream Manager could improve latency and performance. Similarly to the instance we could split it into two threads with a routing thread that makes the routing decisions and a gateway thread that sends out the messages. This would solve the libevent issue we discovered.

Further, reworking Heron to either not rely on tuple sets or to not rely on batching these sets but to directly send it out as soon as ready could be interesting. By addressing the double batching described earlier this might result in a more predictable and possibly more performant system.

For future work, additional benchmarks with higher loads and more complex topologies could lead to other interesting findings. Especially for high throughput workloads, we expect the single threaded Stream Manager to become the bottleneck. A detailed look into the Stream Manager and its buffers could lead to better insight into this problem. Also in our benchmarks we noticed tail latencies which were up to 10 times higher than the median, a closer look into what causes these tail latencies would definitely be interesting.

The instrumentation developed for this thesis gave valuable insight into the inner workings of Heron, but the way these metrics are collected makes it cumbersome to use it for tuning Heron for a specific workflow. A next step would be to integrate these metrics into an existing monitoring system, such as Prometheus [21], or analysis systems, such as Snailtrail [25].

References

[1] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 393–406. ACM, 2015.

[2] Mosharaf Chowdhury and Ion Stoica. Coflow: A networking abstraction for cluster applications. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 31–36. ACM, 2012.

[3] Libevent - an event notification library. https://libevent.org/.

[4] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[6] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[7] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[8] Kostas Tzoumas, Stephan Ewen, and Robert Metzger. High-throughput, low-latency, and exactly-once stream processing with apache flink. *URL https://data-artisans.com/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink*, 2015.

[9] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[10] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[11] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. *HotCloud*, 12:10–10, 2012.

[12] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[13] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Utility and Cloud Computing (UCC), 2014 IEEE/ACM 7th International Conference on*, pages 69–78. IEEE, 2014.

[14] Avrilia Floratou, Ashvin Agrawal, Bill Graham, Sriram Rao, and Karthik Ramasamy. Dhalion: self-regulating stream processing in heron. *Proceedings of the VLDB Endowment*, 10(12):1825–1836, 2017.

[15] Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, 2018. USENIX Association.

[16] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, and Shivakumar Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 577–588. ACM, 2013.

[17] Apache heron. https://github.com/apache/incubator-heron.

[18] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[19] Apache aurora. https://aurora.apache.org/.

[20] Protocol buffers. google's data interchange format. https://code.google.com/apis/protocolbuffers/, 2008.

[21] Prometheus - monitoring system & time series database. https://prometheus.io/.

[22] Hdrhistogram: A high dynamic range histogram. http://hdrhistogram.org/.

[23] Add signal handling to stream manager. https://github.com/apache/incubator-heron/pull/2950.

[24] Reddit in 2015. https://redditblog.com/2015/12/31/reddit-in-2015/.

[25] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. Snailtrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten Semester-, Bachelor- und Master-Arbeit oder anderen Abschlussarbeit (auch der jeweils elektronischen Version).

Die Dozentinnen und Dozenten können auch für andere bei ihnen verfasste schriftliche Arbeiten eine Eigenständigkeitserklärung verlangen.

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und Betreuerinnen der Arbeit.

Titel der Arbeit (in Druckschrift):

Snailtrail for latency-sensitive workloads

Verfasst von (in Druckschrift):

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Fischer	Fabian

Ich bestätige mit meiner Unterschrift:

- Ich habe keine im Merkblatt „Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 9. September 2018	F. Fischer

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.