Scalable Data Analysis (CIS 602-02)

Tools

Dr. David Koop
EC2 Spark Demo
Publish-Subscribe Messaging

• How do you allow lots of data sources to be seen by lots of clients?

• Publish-subscribe: various **producers** publish data and various **consumers** subscribe to receive only the data they care about (e.g. via keywords, categories)

• Kafka: a distributed, partitioned, replicated commit log service
  - Messages guaranteed to be in sent order for a topic
  - Consumers see messages in log order
  - With replication factor N, tolerate up to N-1 server failures

[kafka.apache.org]
Storm

- Streaming system
- Spouts and bolts
- Process tuples
- Topology is a graph of computation where each link represents data flows between nodes
- "at least once" and "at most once" guarantees
Storm Cluster

[http://storm.apache.org/tutorial.html]
Storm Spouts, Bolts, and Topology

[http://storm.apache.org/tutorial.html]
Storm Worker

Each of them could belong to different topologies. A single host may run multiple worker processes, but into a process, each executor is mapped.

As described in [20], a Storm topology is directed graph of the stream. Spouts often pull data from queues, such as Kafka [14], and bolts are an abstraction to represent computation on tweet streams. Spouts and bolts are run as active users in real-time (RTA).

The interest in stream data processing systems includes a flurry of recent systems, including [2, 3, 5, 9, 15, 17, 18]. The initial work about a decade ago (e.g. [6, 7, 10]). The need for highly-scalable stream processing systems has lead to the creation and Heron are presented in Section 6. Finally, Section 8 contains our concluding remarks and points to 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 5.
Spark Streaming

T. Das
Storm vs. Spark Streaming

- Storm: streaming system with micro-batch support
- Spark Streaming: batch system with micro-batch support
Twitter Heron

• Issues with Storm @Twitter:
  - Debugging was hard
  - Storm Worker Architecture:
    • Complex, could run multiple tasks
    • Resource allocation was assumed to be homogeneous
  - Storm Nimbus:
    • Topologies run in isolation -> wastes resources
    • Single point of failure
  - Lack of backpressure
  - Efficiency:
    • Suboptimal Relays
    • Garbage Collection
Heron Architecture

A Heron topology is equivalent to a Storm topology, and bolts do the actual computation. A logical query plan in a Storm application runs on top of a physical execution plan (which is constrained by the actual physical infrastructure available). Storm organizes topologies as a directed acyclic graph of spouts and bolts. Like Storm, Heron runs a Storm API. Thus, the data mode and API for Heron are identical to that of Storm [20].

A key design goal for Heron is to maintain compatibility with the existing topologies and components within Twitter. In this section, we briefly describe our design alternatives and the process of architecting Heron to support them.

### 5.1 Data Model and API

Since the issues discussed in Section 3 are fundamental to Storm, and since the Storm API is already used at Twitter, we had to consider an approach that leverages it. The next option was to consider using another existing open-source system.

Rewriting the existing topologies with a different API would have required much longer development cycles. Also note that there are different libraries that have been developed on top of the Storm API, such as Summingbird [8], and our experience with these systems has been time consuming resulting in a very long migration process. Thus, we concluded that our best option was to rewrite the system from ground-up, reusing and building on some of the existing solutions, such as Apache Samza [2] or Spark Streaming [18]. However, there are a number of issues with respect to these systems.

**Issue 1:** Most of the current open-source systems use an in-memory data model, which is wasteful for the volume of data we process.

**Issue 2:** The current systems are not compatible with Storm's API.

**Issue 3:** Opening up the API to allow for greater degrees of freedom while keeping the data model and API for Heron the same would have been inflexible, and potentially required a fundamental way of changing the architecture. Modify the existing system in such a way that it becomes hard. Changing this existing system in such a way that it is hard to change in the future is not reasonable.

**Issue 4:** However, these systems are not compatible with Storm's API.

**Issue 5:** Fixing them in Storm would have required extensive rewrite of the current form at our scale. In addition, these systems are not compatible with Storm's API and thus, we would have to change other components in our stack. 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.

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

- **At least once:** Each tuple is guaranteed to be processed exactly once, and may contribute to the result of the topology.
- **At most once:** Each tuple is processed at most once, although some tuples may be dropped, and thus may miss being analyzed by the topology.
- **Exactly once:** Each tuple is processed exactly once, and no tuple is processed more than once.
Heron Topology Architecture

Figure 3: Heron Architecture

Figure 4: Heron Topology Architecture

[Figure 3: Heron Architecture]

[Figure 4: Heron Topology Architecture]

[S. Kulkarni et al., 2015]
disadvantages, as the user code is buffered. Once the buffer exceeds a certain threshold, it is delivered to the local SM. If the user logic code program generates an output tuple, it calls the "nextTuple" method to inject data into the topology. The emitted data is then injected into the topology. The task execution thread invokes the "execute" method with the incoming tuple for processing. In the case of a spout, it repeatedly fetches data from the source, and the user logic code invokes the "execute" method. 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 calls the "nextTuple" method to inject data into the topology. The emitted data is then injected into the topology. The task execution thread invokes the "execute" method with the incoming tuple for processing. In the case of a bolt, when tuples arrive, the user logic code invokes the "execute" method to process the input tuple. The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode.

The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode. The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode. The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode. The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode. The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode. The rationale for this design is to prevent a topology from rapidly oscillating between going in and coming out of the backpressure mitigation mode.
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>
<thead>
<tr>
<th>Components</th>
<th>Expt. #1</th>
<th>Expt. #2</th>
<th>Expt. #3</th>
<th>Expt. #4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spout</td>
<td>25</td>
<td>100</td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td>Bolt</td>
<td>25</td>
<td>100</td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td># Storm workers</td>
<td>25</td>
<td>100</td>
<td>200</td>
<td>500</td>
</tr>
<tr>
<td># Heron containers</td>
<td>25</td>
<td>100</td>
<td>200</td>
<td>500</td>
</tr>
</tbody>
</table>

### 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.
EC2 Spark Demo