Hailstorm: Distributed Stream Processing with Exactly Once Semantics

CS240h Final Project, Spring 2014

Thomas Dimson Milind Ganjoo

Stanford University

tdimson@cs.stanford.edu mganjoo@cs.stanford.edu

Abstract

In recent years, *stream processing* has emerged as data analysis technique to handle real-time applications where the latency of Hadoop is inappropriate. Many popular systems, such as Twitter's Storm, provide a rigid platform for performing distributed computations over the network. Storm-like systems typically provide at-least-once processing with state management left to the implementor. We present a novel distributed stream processing framework, *Hailstorm*[1], which provides a platform to perform distributed computation on streams of data in Haskell. By restricting the class of computation to commutative monoids, our system is able to provide exactly-once semantics with little performance loss or added complexity.

1. Introduction

As the Internet has evolved so have user expectations in regards to latency. In one example, Twitter's trending topics feature allows users to see breaking stories within minutes of their emergence. In another, Google Analytics, administrators are able to see detailed demographic information of surfers in real time. The volume and velocity of the data in these systems presents challenges to typical single-machine programs: data does not fit into memory, and latency requirements imply that error recovery has to be automatic and nearly instantaneous.

Like MapReduce [6] and batch processing, frameworks such as Twitter's Storm [4] and LinkedIn's Samza [3] have been created to ease the development of stream processing applications. In these systems, events of interest are pushed into distributed queues from user-facing applications (e.g., Twitter's web site). As the events are popped off the queues, the stream processing framework takes over and transforms the event using a sequence of computations. For example, we might receive Tweets from the queue, split on whitespace and perform a windowed count to determine topics that are currently trending. Similar to MapReduce, developers using these systems write algorithms that operate on individual stream *units* and emit zero or more *messages* to be handled by the next stage in computation. The frameworks distribute the events to clusters running the computation, abstracting away the unreliable nature of the network.

This paper introduces Hailstorm, a stream processing framework in Haskell. Unlike Storm and Samza, Hailstorm mandates that all streaming computations must be both commutative and monoidic. Like Samza, it requires that all events must be initially stored as messages in Apache Kafka [2]. These restrictions allow Hailstorm to make stronger processing guarantees about events: namely, that the each event will be processed *exactly* once in the system. Furthermore, unlike Storm and Samza, state recovery under error conditions is built-in to the framework. We utilize Haskell's purity to guarantee that side-effects of computation are isolated to a single *sink* processor at the end of the computation sequence.

2. Related Work

Hailstorm's technical design is based on that of Apache Storm [4]. Storm is a widely used stream processing framework for the Java Virtual Machine (JVM) allowing developers to upload jobs for continuous processing on a Storm cluster. Developers create a directed acyclic graph of interconnected processing layers called a *topology*. Messages are passed between layers as *tuples*. Tuples originate in a *spout*, which typically reads off of a distributed queue and are passed between layers of *bolts* which perform computation. Each bolt receives a tuple, performs a computation, and emits zero or more tuples to the next layer. Unlike Hailstorm, the bolts have may have side effects to their computation and state management / error recovery is left up to each developer. Accordingly, the system is only able to provide "at least once" guarantees for processing each message in the queue. On component failure, Storm enters a "tu-

[1] `https://github.com/hailstorm-hs/hailstorm`

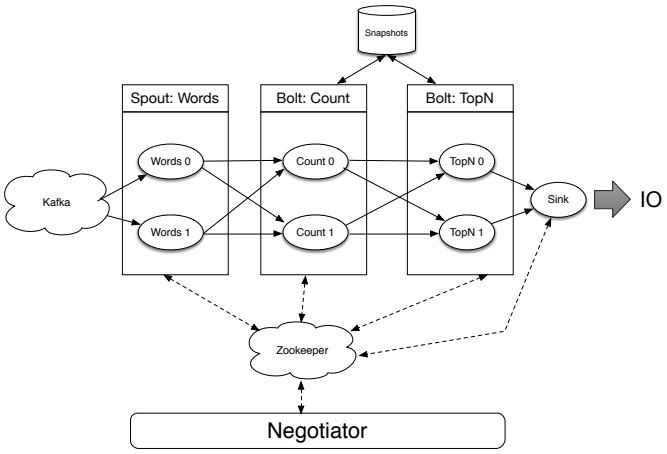


Figure 1: An example Hailstorm topology for word counts

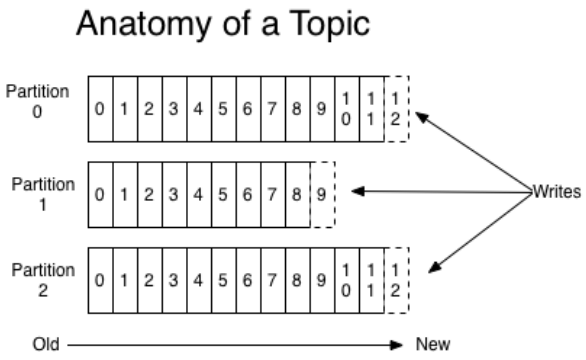


Figure 2: Structure of Kafka partitions

ple replay" state where it re-sends messages from spouts in a topology.

The theoretical underpinings of Hailstorm are inspired by a online essay, "Exactly Once Semantics" [12]. Jackson, a contributor to the Storm framework, describes Kafka log offsets as a vector clock for the system state. This clock allows separate processors to perform synchronized snapshots without locking or direct communication. We further describe the offset clock in Section 3.3.1.

Google's MillWheel system [1] also addresses the issue of exactly once delivery of messages in a stream processing context. Like Storm, messages flow through layers of computation to end up at a final result. MillWheel provides exactly-once semantics by maintaining set of recently processed tuples, discarding those that have recently appeared. Users of MillWheel are required to manually ensure that all computations are idempotent, as system failure induces message re-delivery to the same processor.

3. Hailstorm Overview

Figure 1 shows a complete example of a Hailstorm system used to calculate trending hashtags in real-time using the Twitter firehose. We give a brief overview of the various components and describe them in detail in the upcoming sections.

- *Apache Kafka* is used as the sole queuing mechanism for messages. Messages are consumed off of Kafka *partitions* and then entered into Hailstorm along with their *offset* within the partition.
- *Spouts* are responsible for getting data into Hailstorm. Along with a user-specified conversion function, they consume `ByteStrings` from Kafka and forward them as tuples to the next layer of computation.
- *Bolts* are the fundamental units of computation in Hailstorm. Bolts take a user-specified pure monoidic operation which takes a (state, input-tuple) pair and produces a (state, output-tuple) pair. Bolt state is periodically persisted to the snapshot store. Figure 1 shows multiple layers of bolts.
- *Sinks* are the final stage of Hailstorm processing. Like bolts, sinks take tuples from the previous layer and perform user-specified computation. However, unlike bolts, the computation runs inside the IO monad allowing the user to connect Hailstorm to the real world: databases, web services or even the console.
- *Topologies* are user-specified directed acyclic graphs which describe how bolts, spouts and sinks connect together.
- *Apache Zookeeper* is used as a global service registry for Hailstorm. Processors are registered as into Zookeeper and removed whenever failures occur.
- *The negotiator* in Hailstorm manages the state of a topology: it is responsible for negotiating tuple snapshots and performing error recovery. The negotiator itself maintains no state: if it dies, it can be resumed on any machine with no data loss.

3.1 Apache Kafka

Apache Kafka [2] is a distributed commit log that is used to buffer data between producing systems (e.g., the Twitter Firehose) and Hailstorm topologies. As described in 3.3.1, Hailstorm requires the use of Kafka so it can guarantee exactly-once processing of messaages. Messages are committed to specific *topics*, each with many *partitions*. Within each partition, commits are guaranteed to be linearly ordered according to time, with Kafka providing an offset for each message. Figure 2, from the Kafka website, shows the anatomy of

a single topic as writes get fanned out to different partitions. Consumers, such as Hailstorm spouts, are able to read from individual partitions with a topic and consume mesages. Since the messages are structured in a log, it is possible to "rewind" consumers and have them read messages from earlier points in the log. Kafka also has a configerable replication mechanism and is able to maintain its ordering even in the event of machine failure.

The latest version of the Kafka protocol lacks bindings for Haskell, however the librdkafka [7] library provides up-to-date bindings for C. As part of completing Hailstorm, we created the *Haskakafka*[2] library that exposes librdkafka through Haskell's C FFI. We have since made the bindings available on Hackage for others to use.

3.2 Apache Zookeeper

Apache Zookeeper [5] is a highly-available distributed configuration service, which Hailstorm uses extensively for process registry and synchronization. Zookeeper's data model is roughly analogous to a tree-structured file system, where *nodes* can either be directories or small files. Nodes are either created indefinitely or registered as *ephemeral nodes*, which are automatically deleted when their creator's connection is terminated. Most Zookeeper libraries also implement *watchers* on nodes which allow a program to be asynchronously notified whenever a node or a node's data changes.

When a Hailstorm processor starts up, it immediately registers an ephemeral node with its identifier underneath a Zookeeper directory called `living_processors`. As described in Section 3.6, the negotiator monitors this directory to ascertain the health of the system.

Hailstorm uses the hzk library [13], which exposes the Zookeeper C library to Haskell. The watcher notifications in Hailstorm occur in a seperate (OS) thread, which communicates the value of the change back to the worker thread using an MVar[3].

3.3 Spouts

Spouts[4] are the starting point for any flow of information through the Hailstorm system. Each spout has a one-to-one connection with a Kafka partition.

When specifying the topology, a client provides a pure function that converts a Kafka message to a tuple in a suitable form for processing by downstream processors (see Listing 1). For example, a simple "word count" topology could convert a word from Kafka into a *(word, 1)* tuple to facilitate counting in downstream bolts.

[2] `http://hackage.haskell.org/package/haskakafka`

[3] See `ZKCluster.hs`

[4] See `Spout.hs`

```
data Spout =
  Spout { -- ...
        , convertFn :: BS.ByteString ->
            PayloadTuple
        , -- ...
        }
```

Listing 1: Client interface for a spout

Along with the tuple itself, a spout also sends a (partition, offset) pair corresponding to the origin of the tuple. This forms the `Payload`, used for downstream processors for snapshotting (see 3.4.1).

3.3.1 Clock

We define the notion of a vector *clock* that determines the state of all processors regardless of message ordering. Put simply, the clock is a map of Kafka partition names to offset values. More formally, a clock C will contain an offset $C[p]$ for each Kafka partition p that feeds the topology.

Hailstorm uses clocks to ensure safe error recovery for processors, as outlined in the following sections.

3.4 Bolts

Bolts[5] form the computational portion of the Hailstorm topology. Each bolt maintains an internal state (represented by the type `BoltState`), which is updated as the computation advances. The key characteristic of Hailstorm bolt states is that they are commutative monoids[6]. This allows any state to be represented as a `mappend` of one or more older states. This has important implications for crash recoverability: having a snapshot of an older state means that computations can be replayed from that point forward instead of having to start from scratch.

A subset of the client-provided bolt interface is shown in Listing 2. Incoming tuples are converted to a monoidal `BoltState` to facilitate merging with the existing state. The conversion is performed using `tupleToStateConverter` (a function that converts a key-value pair into a singleton map would be an example). The `mergeFn` performs commutative `mappend`. The tuples themselves are transformed using the new state and `transformTupleFn`) and forwarded downstream.

3.4.1 Updating state with incoming tuples

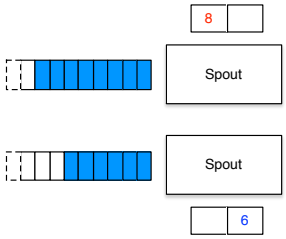
Incoming tuples are always merged into the existing bolt state. Depending on whether the negotiator has posted a desired snapshot clock (see Section 3.6), this could involve either one or two states:

[5] See `Downstream.hs`

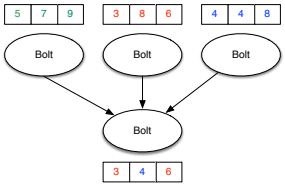
[6] This is not enforced. It is the client's responsibility to provide a `mergeFn` definition that performs a commutative `mappend` between two `BoltState` instances.

```
data Bolt =
  Bolt { -- ...
       , transformTupleFn :: PayloadTuple ->
           BoltState -> PayloadTuple
       , emptyState :: BoltState
       , mergeFn :: BoltState -> BoltState ->
           BoltState
       , tupleToStateConverter :: PayloadTuple
           -> BoltState
       -- ...
       }
```

Listing 2: Client interface for a bolt



(a) LWM for a spout has just one entry: the offset in the associated partition.



(b) LWM for a bolt is calculated as $min(LWM_k)$ for each upstream k.

Figure 3: Calculating LWM for processors.

- When there is no snapshot being requested, incoming tuples are merged into a single state instance. The output tuple from a bolt is calculated using this single state.
- When a bolt receives a negotiator request to perform a snapshot, it splits its current state into state A and B: its current state becomes A and it initializes B to `mempty`.
- As long as the bolt is not eligible to perform a snapshot (see Section 3.4.2 for when this is determined), the bolt maintains two states: pre-snapshot state A and post-snapshot state B. It merges tuples into the appropriate state based on their source partition offsets, and calculates the downstream tuple based on a combination of these two states. More formally, for a desired snapshot clock C, if an incoming tuple originated from partition p at offset o, then the tuple will be merged into state A if $o \leq C[p]$; otherwise, it is merged into B. The downstream output tuple will be calculated from $A + B$.
- When it is time to actually persist its state, the bolt forks a thread to persist pre-snapshot state A. In the main thread, it merges states A and B and merges new tuples into this single state, reverting to the no-snapshot phase.

3.4.2 Low Water Mark

After the snapshot request is received and a bolt bifurcates its state, it must wait till the pre-snapshot state is guaranteed to no longer be affected by incoming tuples. To help determine when this happens, bolts use another piece of information: the *low water mark (LWM)*.

A low water mark LWM_k for a processor k is simply a clock (see Section 3.3.1), where offset $LWM_k[p]$ for partition p is the *lowest* offset seen by any processor upstream to k. It is calculated recursively (as shown in Figure 3), and indicates the *least* amount of progress made for each partition in the entire topology.

To help downstream bolts calculate their LWMs, payloads carry a map of upstream processor names to their respective LWMs, which is updated at each level.

The LWM is used in determining snapshot eligibility as follows: a bolt may only persist its state when its LWM equals the desired snapshot clock in *every* dimension. When that happens, future tuples are guaranteed to originate from offsets greater than ones in the desired snapshot clock, and thus the pre-snapshot state for a bolt will no longer change.

3.4.3 Saving and restoring snapshots

Bolts receive an instance of a `SnapshotStore` typeclass with two functions: `saveSnapshot` and `restoreSnapshot`. When a bolt is eligible to snapshot its state, it forks a thread to perform the save. On boot-up, a bolt tries to restore its state from the provided `SnapshotStore`. If there is nothing saved, it starts from an empty state.

In the current Hailstorm implementation, we provide a `DirSnapshotStore` instance that creates a snapshot in a local directory. Ideally, one would want to provide other instances: one that uses an SQL database, or one that uses a distributed file system for greater fault tolerance.

3.5 Sinks

Sinks are at the bottom-most level of a topology. As the final computation step, they serve the role of gateway to the real world. Sinks often perform actions like printing to console or writing to a database (they are the only processors in the topology that are allowed to execute impure code). The user action is provided by the client in the form of a Pipes `Consumer` that accepts upstream tuples (see Listing 3).

There is no restriction to the type of computation performed on incoming tuples; however, since no snapshots are performed on sinks, non-idempotent operations may have unexpected results.

```
data Sink =
  Sink { -- ...
       , outputConsumer :: Consumer
           PayloadTuple IO ()
       , -- ...
       }
```

Listing 3: Client interface for a sink

3.6 Negotiator

The *negotiator*[7] has full control over all the processors of a Hailstorm topology. The negotiator shares two related roles:

1. Forcing bolts to snapshot with a valid clock
2. Recovering the state of the system if a processor becomes unreachable

Upon registration, the negotiator creates a special ephemeral node for the topology called the *master state*. The negotiator transitions the master state through the deterministic finite automaton shown in Figure 4. Each processor in the topology creates a watcher for the master state, responding quickly to transitions.

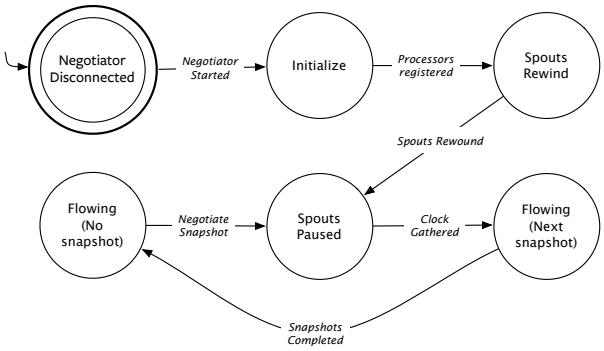


Figure 4: Master state machine for a topology. After initialization, the topology loops between flowing and snapshotting indefinitely

Initially, the topology begins in the `NegotiatorDisconnected` state which is indicated by the absence of a master state node in Zookeeper. When the negotiator boots up, it sets the master state to `Initialize` and waits for all processors to register under living processors (see Section 3.2). Once the expected nodes have been created, the negotiator waits for the bolts to load their snapshots from the snapshot store (see Section 3.4.3). The bolts communicate their snapshot clock their living processor nodes, and negotiator is then able change the master state to `SpoutsRewind`. The spouts rewind to the clock, and pause, writing their position into their corresponding living processor nodes. Finally, the negotiator begins the main *run loop*.

Hailstorm's run loop consists of the negotiator alternating between snapshots and a grace period of data flow. When data is flowing without a desired snapshot clock, the negotiator sets the master state to `Flowing Nothing` and then waits a configerable grace period. When the grace period expires, the negotiator sets the state to `SpoutsPaused` and determines the next snapshot clock from the current spout offsets. When complete, the negotiator sets the master state to `Flowing NextClock` , which the bolts use to determine their snapshot. After the bolt snapshots are complete, the negotiator returns to `Flowing Nothing` and loops.

The run loop can be interrupted by an unreachable processor. When a processor becomes unresponsible, their Zookeeper connection terminates and their corresponding ephemeral node is removed from the living processors directory. The negotiator is asynchronously notified through a watcher, and then sets the master state to `Initialize`. As part of this transition, the negotiator removes the living processor nodes for all processors in the topology. Each processor reads this as a signal to restart, so that the topology can restart in a clean state. Thus the `Initialize` master state is identical to that of the negotiator's initial start.

4. Example Topology

In light of the technical details of Section 3, we return the sample topology of Figure 1. The topology uses Hailstorm to calulate trending hashtags in real-time from Twitter [8]. In a non-distributed setting, the computation would typically use the unix `sort` and `uniq` commands. Hailstorm enables the same principles to scale to datasets far beyond a single computer's memory.

On the producer side, we enqueue messages into Kafka using a Python script. The script samples hashtags from the Twitter's web API and then enqueues them into Kafka as UTF-8 bytestrings .

As data streams into Kafka, Hailstorm streams data out. The spouts take the UTF-8 bytestrings, convert them into Haskell character lists and send them to the next layer of *count* bolts. Our hashing function, the native Haskell string hash, forces a 1:1 mapping betweem hashtag and bolt instance. For example, #love, would always be mapped to *count-0*.

The *count* bolts aggregate individual hashtag occurrences into a running sum. This running sum is stored as a hash map 0 default value, making each addition a monoidic operation. Hashtags are received, aggregated, and emitted with the running count. For example, *count-0* could receive `(#love,1)` and emit `(#love, 101)` to the next layer of *topn* bolts.

[7] See `Negotiator.hs`

[8] See `Sample/WordCountSample.hs`

Each *topn* bolt keeps track of the local top n trending hashtags that have been sent to it. The computation utilizes the Haskell PSQueue library [11], implementing a pure keyed priority queue. Each hashtag is added to the queue, which is then trimmed to the top n entries. The entire queue is sent to the next (and final) layer for processing.

In the sample topology, the *sink* receives the local top n queues and merges them to compute the global top n hashtags. Finally, the top n tags are emitted to the console. The action occurs in the IO Monad and could be modified to update a database or website as appropriate.

5. Implementation Details

5.1 Running Processors

Each of the processors in the topology (spouts, bolts and sinks) can be run independently, as long as they are uniquely identifiable (such as through unique port numbers). The `HailstormCLI.hs` sample executable included with the library allows such behavior, with processors runnable on different threads, cores, or machines.

5.2 Network Processing

Hailstorm utilizes the Haskell Pipes [8] library in place of Lazy I/O. Within a processor, the next layer is modeled as a `Pool` consumer[9] that keeps a connection pool of downstream processor sockets. The `Pool` consumer waits for a Hailstorm `Payload`, hashes it, and then sends it via a network socket `Handle`. Handles themselves are lazily created and maintained within a connection pool. In our initial implementation, messages are serialized using Haskell's `Show` method; we intend to migrate to a more efficient protocol in upcoming versions.

Bolt and spout layers listen for incoming connections and process their messages. After initialization, they instantiate a *mailbox* using the Pipes-Concurrency [9] library and fork a listener thread[10]. The listener thread accepts upstream connections and forks handlers that push incoming tuples into the mailbox. The main processor thread creates a Pipes pipeline that consumes messages from the mailbox, processes them in a pipes and then sends output to a consumer (`Pool` for bolts, IO for sinks).

6. Next Steps

Our Hailstorm implementation is functionally complete, but could use some polishing before a public release.

Currently, the only the `HardcodedTopology` data type conforms to the the `Topology` type-class. Accordingly, a user of Hailstorm has to program the network port and address for each processor into the Hailstorm binary itself. A modification to Hailstorm would register each the network address in the Zookeeper processor registry. The modification would allow processors to be resumed on different machines then they started on.

In that direction, we would like to structure the framework closer to a Hadoop cluster wherein developers upload *jobs* to Hailstorm. We envision developers "uploading" their processors into Zookeeper, with Hailstorm executing the specification. The framework would utilize a package like hint [10] to provide dynamic code execution from Zookeeper.

7. Conclusion

This paper introduces Hailstorm, a Storm-like distributed stream processing framework for Haskell. By restricting our class of computation to commutative monoids and by exploiting Haskell's purity, the system guarantees exactly-once processing of messages without performance loss. Hailstorm maintains these guarantees even in the face of machine failures and an unreliable network. We look forward to developing it further.

References

[1] T. Akidau, A. Balikov, K. Bekiroglu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. In *Very Large Data Bases*, pages 734–746, 2013.

[2] Apache Software Foundation. Apache Kafka, June 2014. URL `http://kafka.apache.org/`.

[3] Apache Software Foundation. Apache Samza, June 2014. URL `http://samza.incubator.apache.org/`.

[4] Apache Software Foundation. Storm, distributed and fault-tolerant realtime computation, June 2014. URL `http://storm.incubator.apache.org/`.

[5] Apache Software Foundation. Apache Zookeeper, June 2014. URL `http://zookeeper.apache.org/`.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. . URL `http://doi.acm.org/10.1145/1327452.1327492`.

[7] M. Edenhill. librdkafka, June 2014. URL `https://github.com/edenhill/librdkafka`.

[8] G. Gonzalez. pipes, June 2014. URL `http://hackage.haskell.org/package/pipes`.

[9] G. Gonzalez. pipes-concurrency, June 2014. URL `http://hackage.haskell.org/package/pipes-concurrency`.

[10] D. Gorin. hint, June 2014. URL `http://hackage.haskell.org/package/hint`.

[11] R. Hinze. psqueue, June 2014. URL `https://hackage.haskell.org/package/PSQueue`.

[9] See `Processor/Pool.hs`

[10] See `Processor/Downstream.hs`

[12] J. Jackson. Exactly once semantics, April 2014. URL https://github.com/jasonjckn/essays/blob/master/exactly_once_semantics.md.

[13] D. Souza. hzk, June 2014. URL http://hackage.haskell.org/package/hzk.