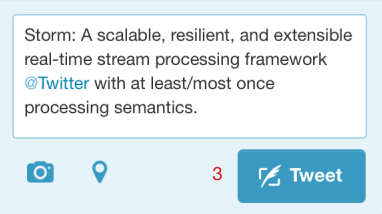
Storm @Twitter

Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel*, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, Dmitriy Ryaboy

@ankitoshniwal, @staneja, @amits, @karthikz, @pateljm, @sanjeevrk, @jason_j, @krishnagade, @Louis_Fumaosong, @jakedonham, @challenger_nik, @saileshmittal, @squarecog

Twitter, Inc., *University of Wisconsin – Madison



ABSTRACT

This paper describes the use of Storm at Twitter. Storm is a real-time fault-tolerant and distributed stream data processing system. Storm is currently being used to run various critical computations in Twitter at scale, and in real-time. This paper describes the architecture of Storm and its methods for distributed scale-out and fault-tolerance. This paper also describes how queries (aka. topologies) are executed in Storm, and presents some operational stories based on running Storm at Twitter. We also present results from an empirical evaluation demonstrating the resilience of Storm in dealing with machine failures. Storm is under active development at Twitter and we also present some potential directions for future work.

1. INTRODUCTION

Many modern data processing environments require processing complex computation on streaming data in real-time. This is particularly true at Twitter where each interaction with a user requires making a number of complex decisions, often based on data that has just been created.

Storm is a real-time distributed stream data processing engine at Twitter that powers the real-time stream data management tasks that are crucial to provide Twitter services. Storm is designed to be:

1. **Scalable**: The operations team needs to easily add or remove nodes from the Storm cluster without disrupting existing data flows through Storm *topologies* (aka. standing queries).
2. **Resilient**: Fault-tolerance is crucial to Storm as it is often deployed on large clusters, and hardware components can fail. The Storm cluster must continue processing existing topologies with a minimal performance impact.
3. **Extensible**: Storm topologies may call arbitrary external functions (e.g. looking up a MySQL service for the social graph), and thus needs a framework that allows extensibility.
4. **Efficient**: Since Storm is used in real-time applications; it must have good performance characteristics. Storm uses a number of techniques, including keeping all its storage and computational data structures in memory.
5. **Easy to Administer**: Since Storm is at that heart of user interactions on Twitter, end-users immediately notice if there are (failure or performance) issues associated with Storm. The operational team needs early warning tools and must be able to quickly point out the source of problems as they arise. Thus, easy-to-use administration tools are not a "nice to have feature," but a critical part of the requirement.

We note that Storm traces its lineage to the rich body of work on stream data processing (e.g. [1, 2, 3, 4]), and borrows heavily from that line of thinking. However a key difference is in bringing all the aspects listed above together in a single system. We also note that while Storm was one of the early stream processing systems, there have been other notable systems including S4 [5], and more recent systems such as MillWheel [6], Samza [7], Spark Streaming [8], and Photon [19]. Stream data processing technology has also been integrated as part of traditional database product pipelines (e.g. [9, 10, 11]).

Many earlier stream data processing systems have led the way in terms of introducing various concepts (e.g. extensibility, scalability, resilience), and we do not claim that these concepts were invented in Storm, but rather recognize that stream processing is quickly becoming a crucial component of a comprehensive data processing solution for enterprises, and Storm

SIGMOD'14, June 22–27, 2014, Snowbird, Utah, USA.

http://dx.doi.org/10.1145/2588555.2595641

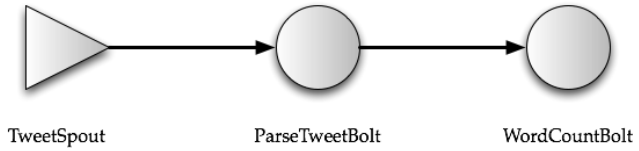


Figure 1: Tweet word count topology

represents one of the early open-source and popular stream processing systems that is in use today.

Storm was initially created by Nathan Marz at BackType, and BackType was acquired by Twitter in 2011. At Twitter, Storm has been improved in several ways, including scaling to a large number of nodes, and reducing the dependency of Storm on Zookeeper. Twitter open-sourced Storm in 2012, and Storm was then picked up by various other organizations. More than 60 companies are either using Storm or experimenting with Storm. Some of the organizations that currently use Storm are: Yahoo!, Groupon, The Weather Channel, Alibaba, Baidu, and Rocket Fuel.

We note that stream processing systems that are in use today are still evolving (including Storm), and will continue to draw from the rich body of research in stream processing; for example, many of these "modern" systems do not support a declarative query language, such as the one proposed in [12]. Thus, the area of stream processing is an active and fast evolving space for research and advanced development.

We also note that there are number of online tutorials for Storm [20, 21] that continue to be valuable resources for the Storm user community.

The move to YARN [23] has also kindled interest in integrating Storm with the Hadoop ecosystem, and a number of resources related to using Storm with Hadoop are now also available (e.g. [21, 22]).

The remainder of this paper is organized as follows: The following section, Section 2, describes the Storm data model and architecture. Section 3 describes how Storm is used at Twitter. Section 3 contains some empirical results and discusses some operational aspects that we have encountered while running Storm at Twitter. Finally, Section 4 contains our conclusions, and points to a few directions for future work.

2. Data Model and Execution Architecture

The basic Storm data processing architecture consists of *streams of tuples* flowing through *topologies*. A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components. Vertices are further divided into two disjoint sets – spouts and bolts. Spouts are tuple sources for the topology. Typical spouts pull data from queues, such as Kafka [13] or Kestrel [14]. On the other hand, bolts process the incoming tuples and pass them to the next set of bolts downstream. Note that a Storm topology can have cycles. From the database systems perspective, one can think of a topology as a directed graph of operators.

Figure 1 shows a simple topology that counts the words occurring in a stream of Tweets and produces these counts every 5 minutes. This topology has one spout (*TweetSpout*) and two bolts (*ParseTweetBolt* and *WordCountBolt*). The TweetSpout may pull tuples from Twitter's Firehose API, and inject new Tweets

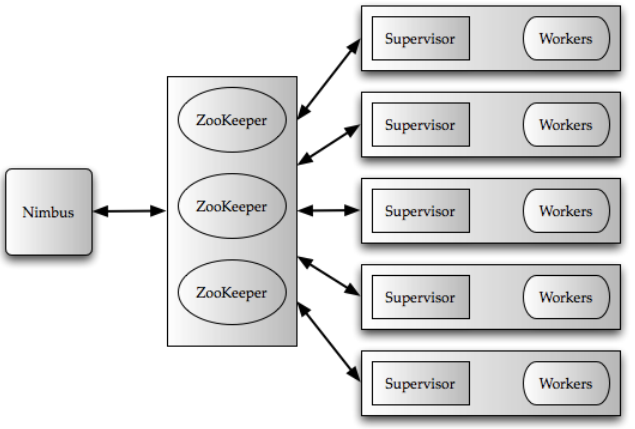


Figure 2: High Level Architecture of Storm

continuously into the topology. The ParseTweetBolt breaks the Tweets into words and emits 2-ary tuples (word, count), one for each word. The WordCountBolt receives these 2-ary tuples and aggregates the counts for each word, and outputs the counts every 5 minutes. After outputting the word counts, it clears the internal counters.

2.1 Storm Overview

Storm runs on a distributed cluster, and at Twitter often on another abstraction such as Mesos [15]. Clients submit topologies to a *master node*, which is called the *Nimbus*. Nimbus is responsible for distributing and coordinating the execution of the topology. The actual work is done on *worker nodes*. Each worker node runs one or more *worker processes*. At any point in time a single machine may have more than one worker processes, but each worker process is mapped to a single topology. Note more than one worker process on the same machine may be executing different part of the same topology. The high level architecture of Storm is shown in Figure 2.

Each worker process runs a JVM, in which it runs one or more *executors*. Executors are made of one or more *tasks*. The actual work for a bolt or a spout is done in the task.

Thus, tasks provide intra-bolt/intra-spout parallelism, and the executors provide intra-topology parallelism. Worker processes serve as containers on the host machines to run Storm topologies.

Note that associated with each spout or bolt is a set of tasks running in a set of executors across machines in a cluster. Data is *shuffled* from a producer spout/bolt to a consumer bolt (both producer and consumer may have multiple tasks). This shuffling is like the exchange operator in parallel databases [16].

Storm supports the following types of partitioning strategies:

1. **Shuffle** grouping, which randomly partitions the tuples.
2. **Fields** grouping, which hashes on a subset of the tuple attributes/fields.
3. **All** grouping, which replicates the entire stream to all the consumer tasks.
4. **Global** grouping, which sends the entire stream to a single bolt.

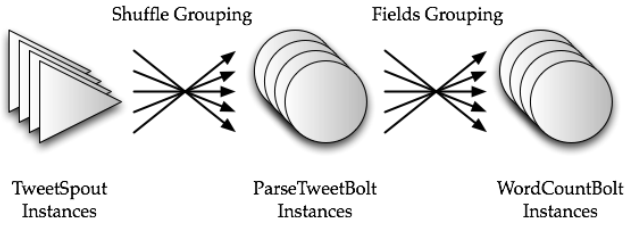


Figure 3: Physical Execution of the Tweet word count topology

5. **Local** grouping, which sends tuples to the consumer bolts in the same executor.

The partitioning strategy is extensible and a topology can define and use its own partitioning strategy.

Each worker node runs a *Supervisor* that communicates with Nimbus. The cluster state is maintained in Zookeeper [17], and Nimbus is responsible for scheduling the topologies on the worker nodes and monitoring the progress of the tuples flowing through the topology. More details about Nimbus is presented below in Section 2.2.1.

Loosely, a topology can be considered as a logical query plan from a database systems perspective. As a part of the topology, the programmer specifies how many instances of each spout and bolt must be spawned. Storm creates these instances and also creates the interconnections for the data flow. For example, the physical execution plan for the Tweet word count topology is shown in Figure 3.

We note that currently, the programmer has to specify the number of instances for each spout and bolt. Part of future work is to automatically pick and dynamically changes this number based on some higher-level objective, such as a target performance objective.

2.2 Storm Internals

In this section, we describe the key components of Storm (shown in Figure 2), and how these components interact with each other.

2.2.1 Nimbus and Zookeeper

Nimbus plays a similar role as the "JobTracker" in Hadoop, and is the touchpoint between the user and the Storm system.

Nimbus is an Apache Thrift service and Storm topology definitions are Thrift objects. To submit a job to the Storm cluster (i.e. to Nimbus), the user describes the topology as a Thrift object and sends that object to Nimbus. With this design, any programming language can be used to create a Storm topology.

A popular method for generating Storm topologies at Twitter is by using Summingbird [18]. Summingbird is a general stream processing abstraction, which provides a separate logical planner that can map to a variety of stream processing and batch processing systems. Summingbird provides a powerful Scala-idiomatic way for programmers to express their computation and constraints. Since Summingbird understands types and relationships between data processing functions (such as associativity), it can perform a number of optimizations. Queries expressed in Summingbird can be automatically translated into Storm topologies. An interesting aspect of Summingbird is that it can also generate a MapReduce job to run on Hadoop. A common use case at Twitter is to use the Storm topology to compute approximate answers in real-time, which are later reconciled with accurate results from the MapReduce execution.

As part of submitting the topology, the user also uploads the user code as a JAR file to Nimbus. Nimbus uses a combination of the local disk(s) and Zookeeper to store state about the topology. Currently the user code is stored on the local disk(s) of the Nimbus machine, and the topology Thrift objects are stored in Zookeeper.

The Supervisors contact Nimbus with a periodic heartbeat protocol, advertising the topologies that they are currently running, and any vacancies that are available to run more topologies. Nimbus keeps track of the topologies that need assignment, and does the match-making between the pending topologies and the Supervisors.

All coordination between Nimbus and the Supervisors is done using Zookeeper. Furthermore, Nimbus and the Supervisor daemons are fail-fast and stateless, and all their state is kept in Zookeeper or on the local disk(s). This design is the key to Storm's resilience. If the Nimbus service fails, then the workers still continue to make forward progress. In addition, the Supervisors restart the workers if they fail.

However, if Nimbus is down, then users cannot submit new topologies. Also, if running topologies experience machine failures, then they cannot be reassigned to different machines until Nimbus is revived. An interesting direction for future work is to address these limitations to make Storm even more resilient and reactive to failures.

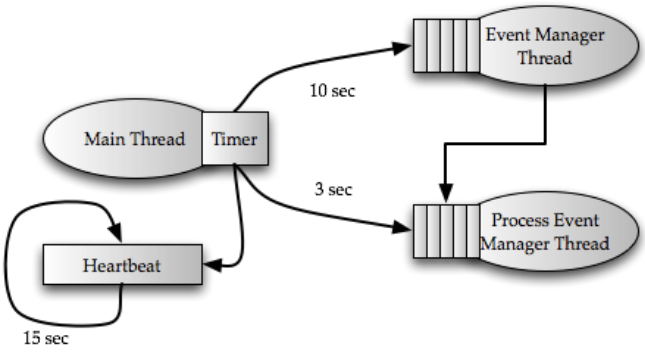


Figure 4: Supervisor architecture

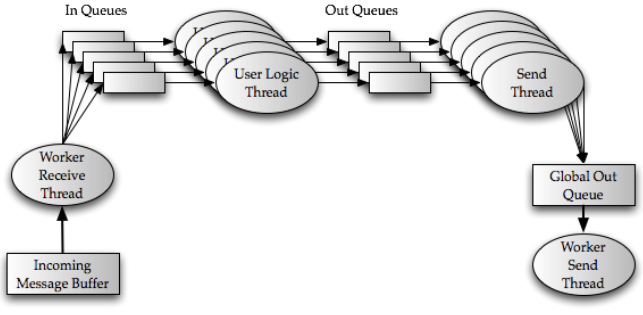


Figure 5. Message flow inside a worker

2.2.2 Supervisor

The supervisor runs on each Storm node. It receives assignments from Nimbus and spawns workers based on the assignment. It also monitors the health of the workers and respawns them if necessary. A high level architecture of the Supervisor is shown in Figure 4. As shown in the figure, the Supervisor spawns three threads. The main thread reads the Storm configuration, initializes the Supervisor's global map, creates a persistent local state in the file system, and schedules recurring timer events. There are three types of events, which are:

1. The **heartbeat** event, which is scheduled to run every 15 seconds, and is runs in the context of the main thread. It reports to Nimbus that the supervisor is alive.
2. The **synchronize supervisor** event, which is executed every 10 seconds in the *event manager thread*. This thread is responsible for managing the changes in the existing assignments. If the changes include addition of new topologies, it downloads the necessary JAR files and libraries, and immediately schedules a synchronize process event.
3. The **synchronize process** event, which runs every 3 seconds under the context of the *process event manager thread*. This thread is responsible for managing worker processes that run a fragment of the topology on the same node as the supervisor. It reads worker heartbeats from the local state and classifies those workers as either *valid*, *timed out*, *not started*, or *disallowed*. A "timed out" worker implies that the worker did not provide a heartbeat in the specified time frame, and is now assumed to be dead. A "not started" worker indicates that it is yet to be started because it belongs to a newly submitted topology, or an existing topology whose worker is being moved to this supervisor. Finally, a "disallowed" worker means that the worker should not be running either because its topology has been killed, or the worker of the topology has been moved to another node.

2.2.3 Workers and Executors

Recall that each worker process runs several executors inside a JVM. These executors are threads within the worker process. Each executor can run several tasks. A task is an instance of a spout or a bolt. A task is strictly bound to an executor because that assignment is currently static. An interesting direction for future work is to allow dynamic reassignment to optimize for some higher-level goal such as load balancing or meeting a Service Level Objective (SLO).

To route incoming and outgoing tuples, each worker process has two dedicated threads – a *worker receive thread* and a *worker send thread*. The worker receive thread listens on a TCP/IP port, and serves as a de-multiplexing point for all the incoming tuples. It examines the tuple destination task identifier and accordingly queues the incoming tuple to the appropriate *in queue* associated with its executor.

Each executor consists of two threads namely the *user logic thread* and the *executor send thread*. The user logic thread takes incoming tuples from the *in queue*, examines the destination task identifier, and then runs the actual task (a spout or bolt instance) for the tuple, and generates output tuple(s). These outgoing tuples are then placed in an *out queue* that is associated with this executor. Next, the executor send thread takes these tuples from the out queue and puts them in a *global transfer queue*. The global transfer queue contains all the outgoing tuples from several executors.

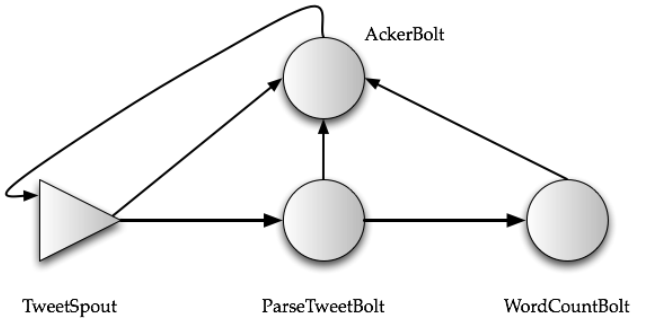


Figure 6. Augmented word count topology

The worker send thread examines each tuple in the global transfer queue and based on its task destination identifier, it sends it to the next worker downstream. For outgoing tuples that are destined for a different task on the same worker, the executor send thread writes the tuple directly into the in queue of the destination task.

The message flow inside workers is shown in Figure 5.

2.3 Processing Semantics

One of the key characteristics of Storm is its ability to provide guarantees about the data that it processes. It provides two types of semantic guarantees – "at least once," and "at most once" semantics.

At least once semantics guarantees that each tuple that is input to the topology will be processed at least once.

With at most once semantics, each tuple is either processed once, or dropped in the case of a failure.

To provide "at least once" semantics, the topology is augmented with an "acker" bolt that tracks the directed acyclic graph of tuples for every tuple that is emitted by a spout. For example, the augmented Tweet word count topology is shown in Figure 6.

Storm attaches a randomly generated 64-bit "message id" to each new tuple that flows through the system. This id is attached to the tuple in the spout that first pulls the tuple from some input source. New tuples can be produced when processing a tuple; e.g. a tuple that contains an entire Tweet is split by a bolt into a set of trending topics, producing one tuple per topic for the input tuple. Such new tuples are assigned a new random 64-bit id, and the list of the tuple ids is also retained in a *provenance tree* that is associated with the output tuple. When a tuple finally leaves the topology, a backflow mechanism is used to acknowledge the tasks that contributed to that output tuple. This backflow mechanism eventually reaches the spout that started the tuple processing in the first place, at which point it can retire the tuple.

A naïve implementation of this mechanism requires keeping track of the lineage for each tuple. This means that for each tuple, its source tuple ids must be retained till the end of the processing for that tuple. Such an implementation can lead to a large memory usage (for the provenance tracking), especially for complex topologies.

To avoid this problem, Storm uses a novel implementation using bitwise XORs. As discussed earlier, when a tuple enters the spout, it is given a 64-bit message id. After the spout processes this tuple, it might emit one or more tuples. These emitted tuples are assigned new message ids. These message ids are XORed and

Figure 7: Storm Visualizations

sent to the acker bolt along with the original tuple message id and a timeout parameter. Thus, the acker bolt keeps track of all the tuples. When the processing of a tuple is completed or acked, its message id as well as its original tuple message id is sent to the acker bolt. The acker bolt locates the original tuple and its XOR checksum. This XOR checksum is again XORed with the acked tuple id. When the XOR checksum goes to zero, the acker bolt sends the final ack to the spout that admitted the tuple. The spout now knows that this tuple has been fully processed.

It is possible that due to failure, some of the XOR checksum will never go to zero. To handle such cases, the spout initially assigns a timeout parameter that is described above. The acker bolt keeps track of this timeout parameter, and if the XOR checksum does not become zero before the timeout, the tuple is considered to have failed.

Note that communication in Storm happens over TCP/IP, which has reliable message delivery, so no tuple is delivered more than once. Consequently, the XORing mechanism works even though XOR is not idempotent.

For at least once semantics, the data source must "hold" a tuple. For the tuple, if the spout received a positive *ack* then it can tell the data source to remove the tuple. If an *ack* or *fail* message does not arrive within a specified time, then the data source will expire the "hold" on the tuple and replay it back in the subsequent iteration. Kestrel queues provide such a behavior. On the other hand, for Kafka queues, the processed tuples (or message offsets) are check pointed in Zookeeper for every spout instance. When a spout instance fails and restarts, it starts processing tuples from the last "checkpoint" state that is recorded in Zookeeper.

At most once semantics implies that the tuples entering the system are either processed at least once, or not at all. Storm achieves at most once semantics when the acking mechanism is disabled for the topology. When acks are disabled, there is no guarantee that a tuple is successfully processed or failed in each stage of the topology, and the processing continues to move forward.

3. Storm in use @ Twitter

In this section, we describe how Storm is used at Twitter. We also present three examples of how we dealt with some operational and deployment issues. Finally, we also present results from an empirical evaluation.

3.1 Operational overview

Storm currently runs on hundreds of servers (spread across multiple datacenters) at Twitter. Several hundreds of topologies run on these clusters some of which run on more than a few hundred nodes. Many terabytes of data flows through the Storm clusters every day, generating several billions of output tuples.

Storm topologies are used by a number of groups inside Twitter, including revenue, user services, search, and content discovery. These topologies are used to do simple things like filtering and aggregating the content of various streams at Twitter (e.g. computing counts), and also for more complex things like running simple machine learning algorithms (e.g. clustering) on stream data.

The topologies range in their complexity and a large number of topologies have fewer than three stages (i.e. the depth of the topology graph is less than three), but one topology has eight

stages. Currently, topologies are isolated on their own machines, and we hope to work on removing this limitation in the future.

Storm is resilient to failures, and continues to work even when Nimbus is down (the workers continue making forward progress). Moreover, if we have to take a machine down for maintenance, then we can do that without affecting the topology. Our p99 latency (i.e. the latency of the 99th percentile response time) for processing a tuple is close to 1ms, and cluster availability is 99.9% over the last 6 months.

3.2 Storm Visualization Operations

A critical part about using Storm in practice is visualizing the Storm operations. Logs from Storm are continuously displayed using a rich visualization developed in-house, some of which are shown in Figure 7. To collect logs, each topology is augmented with a metrics bolt. All the metrics collected at each spout or bolt are sent to this bolt. This bolt in turn writes the metrics to Scribe, which routes the data to a persistent key value store. For each topology, a dashboard is created using this data for visualizing how this topology is behaving.

The rich visualization is critical in assisting with identifying and resolving issues that have caused alarms to be triggered.

The metrics can be broadly classified into system metrics and topology metrics. System metrics shows average CPU utilization, network utilization, per minute garbage collection counts, time spent in garbage collection per minute, and memory usage for the heap. Topology metrics are reported for every bolt and every spout. Spout metrics include the number of tuples emitted per minute, the number of tuple *acks,* the number of *fail* messages per minute, and the latency for processing an entire tuple in the topology. The bolt metrics include the number of tuples executed, the *acks* per minute, the average tuple processing latency, and the average latency to *ack* a specific tuple.

3.3 Operational Stories

In this section we present three Storm-related operational scenarios/stories.

3.3.1 Overloaded Zookeeper

As discussed above, Storm uses Zookeeper to keep track of state information. A recurring issue is how to set up and use Zookeeper in Storm, especially when Zookeeper is also used for other systems at Twitter. We have gone through various considerations about how to use and configure Zookeeper with Storm.

The first configuration that we tried is to use an existing Zookeeper cluster at Twitter that was also being used by many other systems inside Twitter. We quickly exceeded the amount of clients that this Zookeeper cluster could support, which in turn impacted the uptime of other systems that were sharing the same Zookeeper cluster.

Our second configuration of a Storm cluster was identical to the first one, except with dedicated hardware for the Zookeeper cluster. While this significantly improved the number of workers processes and topologies that we could run in our Storm cluster, we quickly hit a limit at around 300 workers per cluster. If we exceeding this number of workers, then we began to witness worker processes being killed and relaunched by the scheduler. This is because for every worker process there is a corresponding *zknode* (Zookeeper node) which must be written to every 15 seconds, otherwise Nimbus deems that the worker is not alive and reschedules that worker onto a new machine.

In our third configuration of a Storm cluster, we changed the Zookeeper hardware and configuration again: We used database class hardware with 6x 500GB SATA Spindles in RAID1+0 on which we stored the Zookeeper transaction log, and a 1x 500GB spindle (no RAID) on which we stored the snapshots. Separating the transaction logs and the snapshots to different disks is strongly recommended in the Zookeeper documentation, and if this recommendation is not followed, the Zookeeper cluster may become unstable. This third configuration scaled to approximately 1200 workers. If we exceeded this number of workers, once again we started to see workers being killed and restarted (as in our second configuration).

We then analyzed the Zookeeper write traffic by parsing the *tcpdump* log from one of the Zookeeper nodes. We discovered that 67% of the writes per second to the Zookeeper quorum was being performed not by the Storm core runtime, but by the Storm library called *KafkaSpout.* KafkaSpout uses Zookeeper to store a small amount of state regarding how much data has been consumed from a Kafka queue. The default configuration of KafkaSpout writes to Zookeeper every 2 seconds per partition, per Storm topology. The partition count of our topics in Kafka ranged between 15 and 150, and we had around 20 topologies in the cluster at that time. (Kafka is a general publisher-subscriber system and has a notion of *topics*. Producers can write about a topic, and consumers can consume data on topics of interest. So, for the purpose of this discussion, a topic is like a queue.)

In our tcpdump sample, we saw 19956 writes in a 60 second window to zknodes that were owned by the KafkaSpout code. Furthermore, we found that 33% of writes to Zookeeper was being performed by the Storm code. Of that fraction, 96% of the traffic was coming from the Storm core that ships with worker processes that write heartbeats to the Zookeeper every 3 seconds by default.

Since we had achieved as much write performance from our Zookeeper cluster as we thought was possible with our current hardware, we decided to significantly reduce the number of writes that we perform to Zookeeper. Thus, for our fourth and the current production configuration of Storm clusters at Twitter, we changed the KafkaSpout code to write its state to a key-value store. We also changed the Storm core to write its heartbeat state to a custom storage system (called "heartbeat daemons") designed specifically for the purpose of storing the Storm heartbeat data. The heartbeat daemon cluster is designed to trade off read consistency in favor of high availability and high write performance. They are horizontally scalable to match the load that is placed on them by the workers running in the Storm core, which now write their heartbeats to the heartbeat daemon cluster.

3.3.2 Storm Overheads

At one point there was some concern that Storm topologies that consumed data from a Kafka queue (i.e. used Kafka in the spouts) were underperforming relative to hand-written Java code that directly used the Kafka client. The concern began when a Storm topology that was consuming from a Kafka queue needed 10 machines in order in order to successfully process the input that was arriving onto the queue at a rate of 300K msgs/sec.

If fewer than 10 machines were used, then the consumption rate of the topology would become lower than the production rate into the queue that the topology consumed. At that point, the topology would no longer be real-time. For this topology, the notion of real-time was that the latency between the initial events represented in an input tuple to the time when the computation

was actually performed on the tuple should be less than five seconds. The specification of the machines used in this case was 2x Intel E5645@2.4Ghz CPUs, 12-physical cores with hyper-threading, 24-hardware threads, 24GB of RAM, and a 500GB SATA disk.

In our first experiment we wrote a Java program that did not use Storm, or any of Storm's streaming computational framework. This program would use the Kafka Java client to consume from the same Kafka cluster and topic as the Storm topology, using just a "for loop" to read messages as fast as possible, and then deserialize the messages. After deserialization, if no other processing was done in this program, then the item would then be garbage collected.

Since this program did not use Storm it didn't support reliable message processing, recovery from machine failure, and it didn't do any repartition of the stream. This program was able to consume input at a rate of 300K msgs/sec, and process data in real-time while running on a single machine with CPU utilization averaging around 700% as reported by the *top* Unix command line tool (with 12-physical cores, the upper bound for the CPU utilization is 1200%).

The second experiment was to write a Storm topology that had a similar amount of logic/functionality as the Java program. We built a simple Storm topology much like the Java program in that all it did was deserialize the input data. We also disabled message reliability support in this experiment. All the JVM processes that executed this Storm topology were co-located on the same machine using Storm's Isolation Scheduler, mimicking the same setup as the Java program. This topology had 10 processes, and 38 threads per process. This topology was also able to consume at the rate of 300K msgs/sec, and process the data in real-time while running on a single machine (i.e. it has the same specifications as above) with a CPU utilization averaging around 660% as reported by *top*. This CPU utilization is marginally lower than the first experiment that did not use Storm.

For the third experiment we took the same topology from experiment two, but now enabled message reliability. This topology needs at least 3 machines in order to consume input at the rate of 300K msgs/sec. Additionally it was configured with 30 JVM processes (10 per machine), and 5 threads per process. The average CPU utilization was 924% as reported by *top*. These experiments give a rough indication of the CPU costs of enabling message reliability relative to the CPU costs associated with deserializing messages (about 3X).

These experiments mitigated the concerns regarding Storm adding significant overhead compared to vanilla Java code that did the same computation, since when both applications provided the same message reliability guarantees, they had roughly the same CPU utilization.

These experiments did bring to light that the Storm CPU costs related to the message reliability mechanism in Storm are non-trivial, and on the same order as the message deserialization costs. We were unable to reproduce the original Storm topology that required 10 machines in a Java program that did not use Storm, as this would involve significant work since this topology had 3 layers of bolts and spouts and repartitioned the stream twice. Reimplementing all this functionality without Storm would require too much time. The extra machines needed could be explained by the overhead of the business logic within this topology, and/or the deserialization and the serialization costs that are incurred when a tuple is sent over the network because the stream needed to be repartitioned.

3.3.3 Max Spout Tuning

Storm topologies have a *max spout pending* parameter. The max spout pending value for a topology can be configured via the "topology.max.spout.pending" setting in the topology configuration yaml file. This value puts a limit on how many tuples can be in flight, i.e. have not yet been acked or failed, in a Storm topology at any point of time. The need for this parameter comes from the fact that Storm uses ZeroMQ [25] to dispatch tuples from one task to another task. If the consumer side of ZeroMQ is unable to keep up with the tuple rate, then the ZeroMQ queue starts to build up. Eventually tuples timeout at the spout and get replayed to the topology thus adding more pressure on the queues. To avoid this pathological failure case, Storm allows the user to put a limit on the number of tuples that are in flight in the topology. This limit takes effect on a per spout task basis and not on a topology level. For cases when the spouts are unreliable, i.e. they don't emit a message id in their tuples, this value has no effect.

One of the problems that Storm users continually face is in coming up with the right value for this max spout pending parameter. A very small value can easily starve the topology and a sufficiently large value can overload the topology with a huge number of tuples to the extent of causing failures and replays. Users have to go through several iterations of topology deployments with different max spout pending values to find the value that works best for them.

To alleviate this problem, at Twitter we have implemented an auto-tuning algorithm for the max spout pending value which adjusts the value periodically to achieve maximum throughput in the topology. The throughput in this case is measured by how much we can advance the progress of the spout and not necessarily by how many more tuples we can push into or through the topology. The algorithm works for the Kafka and Kestrel spouts, which have been augmented to track and report the progress they make over time.

The algorithm works as follows:

a) The spout tasks keep track of a metric called "progress." This metric is an indicator of how much data has been successfully processed for this spout task. For the Kafka spout, this metric is measured by looking at the offset in the Kafka log that is deemed as "committed," i.e. the offset before which all the data has been successfully processed and will never be replayed back. For the Kestrel spout, this metric is measured by counting the number of acks that have been received from the Storm topology. Note that we cannot use the number of acks received as the progress metric for the Kafka Spout because in its implementation, tuples that have been acked but not yet committed could still be replayed.

b) We have a pluggable implementation of the max spout parameter "tuner" class that does auto-tuning of the max spout pending values. The two APIs that the default implementation support are:

- *void autoTune(long deltaProgress*), which tunes the max spout pending value using the progress made between the last call to autoTune()
- *long get()*, which returns the tuned max spout pending value.

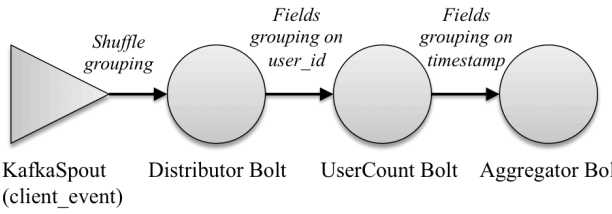


Figure 8: Sample topology used in the experiments

c) Every "*t*" seconds (in our case the default value for t is 120 seconds), the spout calls *autoTune* and provides it the progress that the spout has made in the last *t* seconds.

d) The tuner class records the last "*action*" that it took, and given the current progress value what *action*s it could take next. The *action* value affects the max spout pending value, and the possible values are: *Increase*, *Decrease,* or *No Change*. The action tagged as *Increase* moves the max spout pending value up by 25%. A *Decrease* action reduces the max spout pending value by *Max (25%, (last delta progress - current delta progress)/last delta progress * 100)*%. A *No Change* action indicates that the max spout pending parameter should remain the same as the current value.

The autoTune function has a state machine to determine the next transition that is should make. This state machine transition is described next:

- If the last action is equal to *No Change*, then
  (i) If this is the first time that auto tuning has been invoked, then set action to *Increase,* and increase the max spout pending value.
  (ii) If the last delta progress was higher than the current delta progress, then set action to *Decrease* and decrease max spout pending.
  (iii) If the last delta progress is lower than the current delta progress, then set action to *Increase* and increase max spout pending.
  (iv) If the last delta progress is similar to the current delta progress, then set action to *No Change* and increment a counter by 1, which states how many consecutive turns we have spent in this *No Change* state. If that counter is equal to 5 then set action to *Increase,* and increase the max spout pending value.
- If the last action is equal to *Increase*, then
  (i) If the last delta progress was higher than the current delta progress, then set the action to *Decrease,* and decrease the max spout pending value.
  (ii) If the last delta progress is lower than the current delta progress, then set the action to *Increase*, and increase the max spout pending value.
  (iii) If the last delta progress is similar to the current delta progress, then set the action to *No Change*, and restore the max spout pending value to the value that it had before the last increase was made.
- If the last action is equal to *Decrease*, then
  (i) If the last delta progress is lower than the current delta progress, then set the action to *Increase*, and increase the max spout pending value.
  (ii) For any other case, set the action to *No Change*.

3.4 Empirical Evaluation

In this section we present results from an empirical evaluation that was conducted for this paper. The goal of this empirical evaluation is to examine the resiliency of Storm and efficiency when faced with machine failures.

For this experiment, we created the sample topology that is shown below, and ran it with "at least once" semantics (see Section 2.3). This topology was constructed primarily for this empirical evaluation, and should not be construed as being the representative topology for Twitter Storm workloads.

For simplicity, in Figure 8, we do not show the acker bolt.

As can be seen in Figure 8, this topology has one spout. This spout is a Kafka spout for a "client_event" feed. Tuples from the spout are shuffle grouped to a Distributor bolt, which partitions the data on an attribute/field called "user_id." The UserCount bolt computes the number of unique users for various events, such as "following," "unfollowing," "viewing a tweet," and other events from mobile and web clients. These counts are computed every second (i.e. a 1 Hertz rate). These counts are partitioned on the timestamp attribute/field and sent to the next (Aggregator) bolt. The aggregator bolt aggregates all the counts that it has received.

3.4.1 Setup

For this experiment, we provisioned 16 physical machines. The initial number of tasks for each component in the topology is listed below:

Component	# tasks
Spout	200
DistributorBolt	200
UserCountBold	300
AggregatorBolt	20

The total number of workers was set to 50 and remained at 50 throughout the experiment. We started the topology on 16 machines. Then, we waited for about 15 minutes and removed/killed three machines, and repeated this step three more times. This experimental setup is summarized below:

Time (relative to the start of the experiment)	# machines	# workers	Approximate #workers/machine
0 minutes	16	50	3
+15 minutes	13	50	4
+30 minutes	10	50	5
+45 minutes	7	50	7
+60 minutes	4	50	12

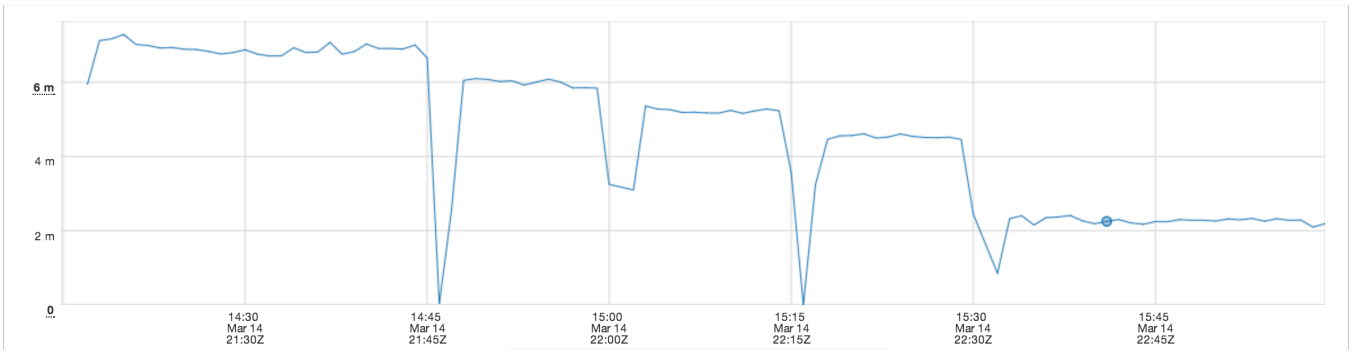


Figure 9: Throughput Measurements

We continually monitored the throughput (# tuples processed by the topology/minute), and the average end-to-end latency (per minute) to process a tuple in the topology. The throughput is measured as the number of tuples acked per minute (in the acker bolt). These results are reported below.

3.4.2 Results

We first report the stable average throughput and latencies below.

Time Window (relative to the start of the experiment)	# machines	Average throughput/ minute (millions)	Average latency/minute (milliseconds)
0-15 minutes	16	6.8	7.8
15-30 minutes	13	5.8	12
30-45 minutes	10	5.2	17
45-60 minutes	7	4.5	25
60-75 minutes	4	2.2	45

The throughput and latency graphs for this experiment, as seen from the visualizer (see Section 3.2), are shown in Figures 9 and 10 respectively.

As can be seen in Figure 9, there is a temporary spike whenever we remove a group of machines, but the system recovers quickly. Also, notice how the throughput drops every 15 minutes, which is expected as the same topology is running on fewer machines. As can be seen in the figure, the throughput stabilizes fairly quickly in each 15 minute window.

Figure 10 shows the latency graph for this experiment, and as expected the latency increases every time a group of machines is removed. Notice how in the first few 15 minute periods, the spikes in the latency graph are small, but in the last two windows (when the resources are much tighter) the spikes are higher; but, as can be seen, the system stabilizes fairly quickly in all cases.

Overall, as can be seen in this experiment, Storm is resilient to machine failures, and efficient in stabilizing the performance following a machine failure event.

4. Conclusions and Future Work

Storm is a critical infrastructure at Twitter that powers many of the real-time data-driven decisions that are made at Twitter. The use of Storm at Twitter is expanding rapidly, and raises a number of potentially interesting directions for future work. These include automatically optimizing the topology (intra-bolt parallelism and the packaging of tasks in executors) statically, and re-optimizing dynamically at runtime. We also want to explore adding exact-once semantics (similar to Trident [24]), without incurring a big performance impact. In addition, we want to improve the visualization tools, improve the reliability of certain parts (e.g. move the state stored in local disk on Nimbus to a more fault-tolerant system like HDFS), provide a better integration of Storm with Hadoop, and potentially use Storm to monitor, react, and adapt itself to improve the configuration of running topologies. Another interesting direction for future work is to support a declarative query paradigm for Storm that still allows easy extensibility.

5. Acknowledgements

The Storm project was started at BackType by Nathan Marz and contributed to, maintained, run, and debugged by countless other members of the data infrastructure team at Twitter. We thank all of these contributors, as this paper would not be possible without their help and cooperation.

6. REFERENCES

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

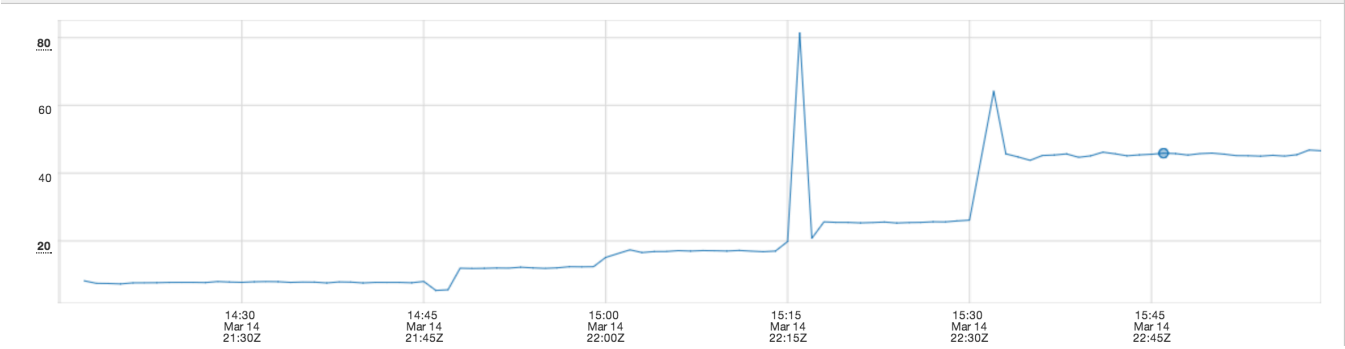


Figure 10: Latency Measurements

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

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

[4] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, Stanley B. Zdonik: The Design of the Borealis Stream Processing Engine. CIDR 2005: 277-289

[5] S4 Distributed stream computing platform. http://incubator.apache.org/s4/

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

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

[8] Spark Streaming. http://spark.incubator.apache.org/docs/latest/streaming-programming-guide.html

[9] Mohamed H. Ali, Badrish Chandramouli, Jonathan Goldstein, Roman Schindlauer: The extensibility framework in Microsoft StreamInsight. ICDE 2011: 1242-1253

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

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

[12] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Ugur Çetintemel, Mitch Cherniack, Richard Tibbetts, Stanley B. Zdonik: Towards a streaming SQL standard. PVLDB 1(2): 1379-1390 (2008)

[13] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. SIGMOD Workshop on Networking Meets Databases, 2011.

[14] Kestrel: A simple, distributed message queue system. http://robey.github.com/kestrel

[15] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. In NSDI, 2011.

[16] Goetz Graefe: Encapsulation of Parallelism in the Volcano Query Processing System. SIGMOD Conference 1990: 102-111

[17] Apache Zookeeper. http://zookeeper.apache.org/

[18] Summingbird. https://github.com/Twitter/summingbird

[19] Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, Ashish Gupta, Haifeng Jiang, Tianhao Qiu, Alexey Reznichenko, Deomid Ryabkov, Manpreet Singh, Shivakumar Venkataraman: Photon: fault-tolerant and scalable joining of continuous data streams. SIGMOD Conference 2013: 577-588

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

[21] Storm, Stream Data Processing: http://hortonworks.com/labs/storm/

[22] Apache Storm: http://hortonworks.com/hadoop/storm/

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

[24] Nathan Marz: Trident API Overview. https://github.com/nathanmarz/storm/wiki/Trident-API-Overview

[25] ZeroMQ: http://zeromq.org/