CTDS Report: Apache Storm

What is Storm?

Apache Storm is an open source framework used to process streams of data on multiple computing nodes. It is written in Clojure (a dialect of Lisp) and has been in development since September 2011. Taken over by Apache in February 2014, Storm is now developed as a Top-Level-Project.

Unlike many already existing multi-node computing frameworks, Storm is specifically created to process data as a stream. Furthermore, the computing job which Storm executes, a so-called Topology, doesn't necessarily have to terminate at some point. Therein lie two of the main differences to the otherwise quite similar and more well-known MapReduce-Job, which essentially processes one big batch of input data. Storm on the other hand never stops accepting input data and keeps applying its predefined operations to the incoming data tuples until its services are no longer needed.

A Storm cluster is also very resilient. Failures of single nodes go by virtually unnoticed due to Storm's ability to dynamically redistribute unoccupied tasks and even restart crashed nodes.

Basic Architecture

To understand the functionality of Apache Storm, it is important to first get a grasp of the underlying principles of a multi-node computing system. The following paragraphs will cover the basic building blocks of Storm and, more importantly, how they all work together.

Pre-Topology Level: Nodes and Clusters

As mentioned earlier, Storm aims to distribute workload among all nodes in the so-called Cluster. In Storm, a node can either take the role of the **Nimbus**, or it can become a **worker node** (that is, run a **Supervisor** process). It is important to note, however, that the node itself is not automatically bound to a specific role: On startup, Storm assigns a daemon to each node, and it is this daemon that separates the Nimbus from the ”common workers”.

The Nimbus process is basically the control center of the entire Cluster. It is aware of all the tasks which the Topology runs and assigns these tasks to the nodes running a worker daemon. Additionally, the Nimbus contains the code for every one of said tasks. Whenever it assigns a task to a worker, the worker is provided with the code snippet that it is supposed to execute. Another very important functionality of the Nimbus is load balancing: The Nimbus is constantly provided with feedback concerning the effectiveness of its workers. If a worker is unable to efficiently execute its assigned tasks, for example due to it being run on weaker hardware than other nodes, the Nimbus tries to alleviate pressure on said weak node and assign some tasks to another, less busy worker in order to optimize performance.

It is important to note that there is only a single Nimbus in every Cluster. This may of course seem counterintuitive at a first glance: If a Cluster contains hundreds or even thousands of worker nodes, this one control node might easily collapse under the sheer pressure that the task of managing every single worker entails. In reality however, the Nimbus utilizes another type of node which exists outside of the Cluster itself, which greatly reduces the processing power required for the Nimbus. But these so-called **Zookeepers** deserve a section of their own and will be discussed later.

While the Nimbus handles these more ”intellectual” tasks like balancing work load, the

worker processes, also called **Supervisors** are, even if their name suggests otherwise, almost the exact opposite. A Supervisor knows nothing of the world surrounding it. It just receives tasks and the code to run them from the Nimbus and executes these tasks until the Nimbus commands them otherwise.

As mentioned earlier, the Nimbus is aware of certain state informations about its Supervisors. In an effort to keep the performance requirements of the Nimbus as low as possible, each Supervisor periodically provides this information to the Zookeeper nodes, where it can easily be accessed by the Nimbus.

Zookeeper Nodes

ZooKeeper is another project by Apache. Much like Storm, Zookeeper is generally run on a cluster of nodes. This is, however, not just unnecessary, but actually hinders performance if used in conjuction with Storm, which is why a Storm-Cluster's Zookeeper system often consists of very few or even a single node (this is due to certain limitations of the Zookeeper architecture - more on that later). The purpose of the Zookeeper Cluster is to provide a hierarchical namespace of data registers which is shared among all Zookeeper server nodes. This allows for very fast and reliable storage and exchange of small data.

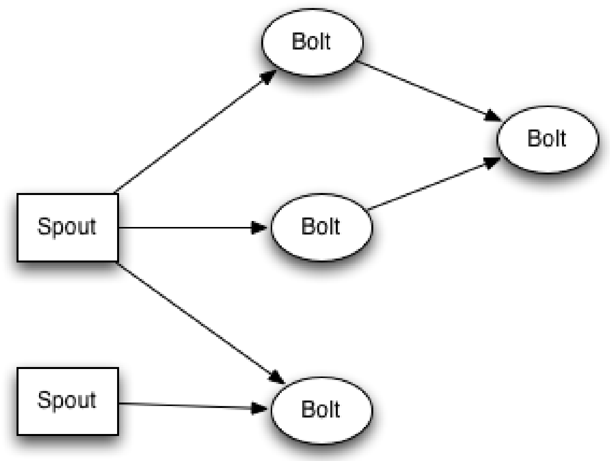
This hierarchical namespace is in many ways quite similar to a file system as we all know it: There is one root node, and every node in the namespace (These nodes are called **znodes**, not to be confused with server nodes, which are actual devices/VMs) is a child node of another znode and can itself have child nodes linking to it, and all these nodes are uniquely identified by an absolute path. But there are also some major differences to a standard file system, the most important of which is the fact that *every* znode, regardless of it being a file or directory, may have data associated with it. Furthermore, unlike in a normal file system, all data is stored in memory, which is why Zookeeper is able to provide disproportionately fast access to its data.

As mentioned earlier, data is transferred to and from the Zookeeper Cluster via a connection to a Zookeeper server node. In order to log its current status information on the Zookeeper system, a Supervisor needs to establish a TCP connection (which is never closed: It would be unreasonable to open a new connection every time since read- and write-requests tend to come in periodically and at a rather high frequency) to a server node and request permission to write into the Zookeeper's file system. This is where one of the weaknesses of Zookeeper becomes evident: Every write-request is passed to *all* the other server nodes, and they must reach consensus before the client is given permission to write (This is necessary to in order to prevent inconsistencies in the data stored on each server). Read-requests however do not need to go through any of this; they are handled locally by the connected server node.

Thus, it becomes evident why Storm is better off using as small a Zookeeper Cluster as possible: All nodes except one periodically send write-requests. Lots of them. And the data provided by the Supervisors is tiny, and therefore easily manageable by a single server node.

Topologies

Now that we've covered the underlying structure with which Storm works, it is time to clarify how the actual work is done. In Storm, all computational tasks are essentially nodes, this time in a graph which is called a **Topology**. This concept is fairly simple: A Topology is an unweighted, directed graph consisting of **Spouts** and **Bolts**. These nodes are then connected by data streams from one node to the other, as is illustrated in the following figure from the Storm website:



Note that every node can potentially be connected to a multitude of Bolts, both through incoming and outgoing streams (Spouts are the exception here. This is due to the different role which the Spouts play in a Topology and will be explained in the following paragraphs)

Of Spouts, Bolts and everything in between

Of course, the Cluster and the Topology aren't just two separate concepts working independently from each other. There is actually a connection between them, and it relies on a type of process mentioned earlier: The *Supervisor*. Although the deeper functionality is much more complex and will be explained later, for now it is important to know that a Supervisor runs a multitude of threads which in turn execute the calculations for the *Components*, which are, dependent on their role, called *Spouts* and *Bolts*.

As mentioned earlier, Storm was created to process streams of incoming data, and it is therefore reasonable to continue our exploration of the Cluster at the very point where the data enters it as well.

Spouts

In Storm, a Spout is the type of node in a Topology which emits one or more streams of data to every node connected to it. This data is usually provided to the Spout by an external service - a good example would be an API from any website which provides data in the form of streams.

It is important that every (physical) device on which a *Supervisor* is run actually has access to the original source of data. This is due to the fact that the developer working with Storm has no control over the location where the Spouts and Bolts are actually run; This assignment is done by the Nimbus. It should therefore be assumed that *every* physical device in the Cluster could potentially be selected to have one or more Spouts running on it. In the case of the original data source being a website, this is of course an almost negligible limitation. But if the data originates from a certain process running on a separate machine, the developer has to make the data available to the Spouts by implementing a server.

When talking about Spouts, there is an important distinction to make between *reliable* and *unreliable* Spouts. Imagine this like the difference between TCP and UDP: An unreliable Spout simply outputs data without requiring confirmation of any sort; a simple fire-and-forget implementation. A reliable Spout on the other hand provides the tools to check not only if the sent data tuple makes it to its recipient, but if the data tuple has actually completed its entire journey through the Topology. The two methods which a reliable Spout provides to that end are called *ack* and *fail*, which are called whenever a tuple emitted from the Spout successfully completes its journey or fails along the road.

Probably the most important Method of Spouts independent of their reliability is called *nextTuple*. Whenever nextTuple is called on a Spout, it emits the next data tuple from its internal buffer onto every stream it is connected to.

Bolts

Unlike the Spouts, whose most difficult computational task is the acquisition of data, the Bolts are the actual workhorse of the Topology. Every Bolt needs at least some kind of data input, and this is done by *subscribing* to a data stream, which may originate from either a Spout or a Bolt.

While the most central method of the Spout is nextTuple, the method which is most often called in Bolts is the *execute* method. The execute method takes a tuple of incoming data and applies the appropriate processing step to it. It is important to note that while a Bolt's execute Method could technically be extremely powerful (the execute method is of course implemented by the developer in his/her language of choice), it is a common practice to split all aspects of the computational task which the Topology shall accomplish into adequately small steps, each of which has an own class of Bolts executing it. This is mainly due to the strong limitation that execute is called for every single data tuple separately, thereby requiring the user to limit the functionality of each Bolt in such a way that it can treat every tuple independently.

As mentioned before, Bolts too can emit streams of data. In order to do so, each processed data tuple is handed to an instance of the OutputCollector class which not only provides methods to emit tuples to streams but also an *ack* and *fail* method to indicate if said tuple has been successfully processed.

Streams and Grouping

In Storm, the streams are used to transfer data between nodes, and are defined as unboundes sequences of data tuples. By default, these data tuples can of course contain the primitive data types as well as Strings and byte arrays. Furthermore, the developer can specify a scheme that declares certain segments of each tuple as fields (Just like for example in a TCP packet, there are fields for ports, sequence number etc). It is also possible for the user to create their own serializable data type so that they can be used within Storm tuples. Each stream is identified by an id given to it upon declaration.

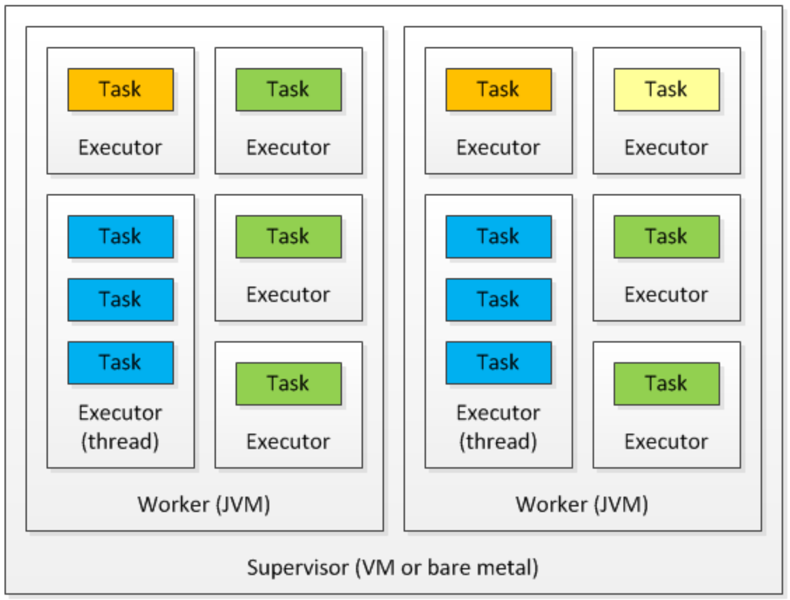
In order to understand stream grouping, one first needs to get a graps on another concept called a *task*. In Storm, a task performs the actual processing of data. While all of the aforementioned information about Spouts and Bolts holds true, it is now necessary to realize that a *Component* itself is not directly executed by a Supervisor: It is rather the *task* which makes up the Component that is run inside a thread on the Supervisor, and the developer is able to specify the number of tasks of each Component. There is still more to say about this topic, but for now, just keep in mind that when the developer programs their Spouts and Bolts, they actually define a task for each Component which emit tuples (for Spouts) or process incoming tuples (for Bolts).

Basically, stream grouping defines how data tuples reaching a Bolt are distributed across the tasks which make up said Bolt. There is a multitude of groupings already implemented in Storm: There is the straight-forward shuffle grouping, which sends each incoming tuple to a random task of the receiving bolt. Another important one is Fields grouping, which routes the tuples to the tasks based on the tuple's content. It is even possible for the producer of the data tuples on a stream to specify to which task of the recipient each tuple is sent.

Arranging Tasks

The last few paragraphs introduced a lot of terminology and concepts which may be quite confusing at a first glance. The differences between them however, while sometimes not very obvious, are important enough to be looked at in more detail, and some concept are also worth reiterating at this point.

First of all, it should be clarified how exactly the tasks fit into the whole picture using the following figure:



We can see that the whole thing depicts a Supervisor daemon which is run on a node in the Cluster, thereby identifying that node as a worker node. The Supervisor then can run one or more Worker processes, each of which is running in its own JVM.

Now comes the interesting part: As mentioned in the last chapter, each Component (Spout or Bolt) has a certain amount of Tasks specified (by the developer) which execute said Component's computations. It is worth noting now, however, that a Task itself is not a thread, but instead is run inside an *Executor* thread. While an Executor can run an arbitrary number of tasks, it is generally recommended to have each task running on its own Executor.

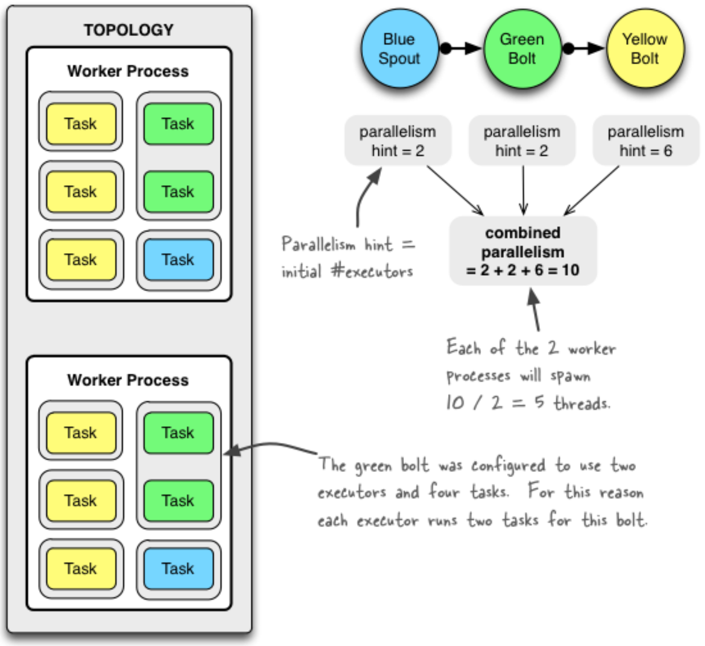
In this illustration, the colour of each Task specifies to which Component it belongs; In this case, the blue Tasks might belong to the Spout called BlueSpout, the green Tasks to the Bolt GreenBolt etc. As a clarification: Every Task of a certain Component runs the exact same code. This is the code specified by the developer, for example in the *execute*-Method of a Bolt.

Another important thing to note is that no Executor runs Tasks from more than one Component.

Parallelism and grouping Tasks into Executors

So far, we have mainly covered the main aspects of the Supervisors - running a certain number of workers, specifying Components and on how many Tasks they run their computations - but the actual functionality of the Nimbus pretty much remained a mystery. The observant reader however might have noticed that especially in the topics covered in these last few paragraphs, the Nimbus has been very active behind the scenes: All kinds of administrative work like assigning Workers to Supervisors, grouping Tasks into Executors and Executors into Workers is actually done by the Nimbus.

Assignment of the Workers is rather straight forward; the Nimbus tends to distribute these evenly across its Supervisors and make changes if necessary. For the rest, there is one last important concept called *Parallelism hint*, which is best explained using another illustration from Storm's website:


TOPOLOGY
Worker Process
Task
Worker Process
Task
Blue Spout
Green Bolt
Yellow Bolt
parallelism hint = 2
parallelism hint = 2
parallelism hint = 6
combined parallelism = 2 + 2 + 6 = 10
Parallelism hint = initial #executors
Each of the 2 worker processes will spawn 10 / 2 = 5 threads.
The green bolt was configured to use two executors and four tasks. For this reason each executor runs two tasks for this bolt.

Not explicitly stated is the fact that the blue Spout has been specified to be run on two Tasks, the green Bolt on four and the yellow Bolt on six. By also defining a parallelism hint, the Nimbus now has all the information it needs to evenly distribute the tasks among their respective Executors, and assign these Executors to the Worker processes.