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.

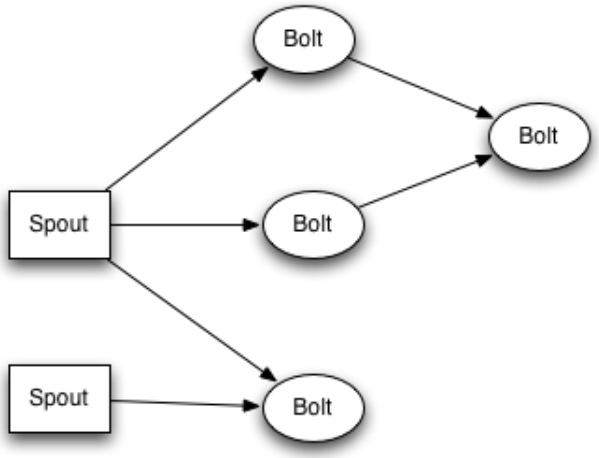
As mentioned earlier, data is transferred to and from the Zookeeper Cluster via a connection to a Zookeeper server node. In order to log its current status information on the Zookeeper system, a Supervisor needs to establish a TCP connection (which is never closed: It would be unreasonable to open a 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 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 actually makes it to its recipient, but if the data tuple has completed its entire journey through the Topology. The two methods which a 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.