1. Architecture Design
   1.1. Overview

   Generally, our Crane's architecture follows that of Storm. **Nimbus** (the master daemon) takes in client's application configuration and is responsible for distributing the tasks to the machines, and also handling machine failures. **Supervisor** (the slave daemon) manages **nodes** (**spout** or **bolt**) according to nimbus's instructions. During application execution, **tuples** flow through the node **topology** and end at **sink** nodes.

   1.2. Fault-tolerance for Tuples

   For terminology clarification, a **message** is the entire entity generated from the spout and flowing through the topology, whereas a **tuple** just lives between two nodes. To guarantee reliable message execution, the spout keeps record of every outgoing message until it's sure that the message is done. This is implemented by letting every relevant node **acknowledge** to the spout after its execution. Specifically, the completeness of tuples' path is mostly equivalent to make sure that once a tuple is emitted from the upstream node, it must be handled by the downstream node later. So we can leverage **XOR** operation on ID's of all the outgoing and incoming tuples related to a message to watch its processing, and when the aggregated XOR is 0, it's done.

   1.3. Task Assignment and Machine Failures

   Nimbus has two views of the whole system. One is the **physical** machines in the cluster maintained by gossiper failure detector. The other one is the **logical** topology it needs to map to some alive machines for the client. To start an application, nimbus parses the configuration and via **RPC** it instructs the supervisors of the chosen machines to start their assigned nodes. To handle a machine failure, nimbus reassigns the nodes to a **substitute** machine and **updates** the nodes' parents' configuration of children's IP's. If the parent machine also fails, it can be **recursively** handled, whose configuration has already included the latest children IP's. If the node's children also fail, the node will eventually know the new children when they update its configuration. So in theory, it can handle much **more simultaneous failures** actually.

2. Programming Framework
   2.1. Topology

   For a client to submit an application job, all he/she needs to write is a configuration file. In this file, the logical machines and nodes are organized **hierarchically**. Almost everything is configurable except the machine IP, which is assigned dynamically depending on the cluster status. The essential part is the setting of executor and function for each bolt.

   2.2. Executor

   Executor is the **generic** role of a bolt, e.g., **filter**, **transform**, **join**, and **reduce**.

   2.3. Function

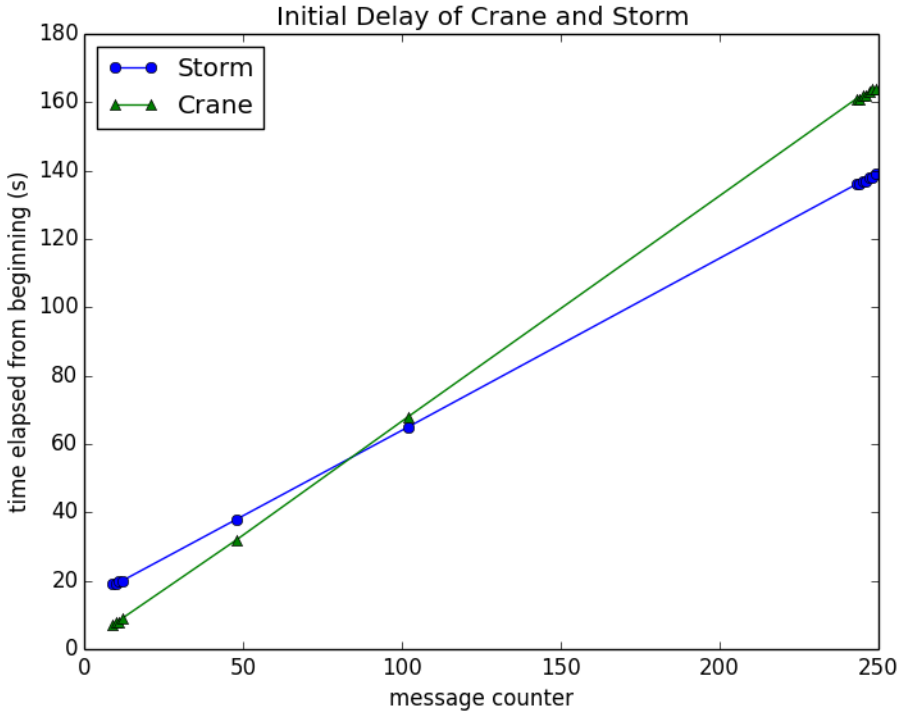
   Function is the **user-defined** criteria or rule determines how executor should deal with the tuples. For life saving, we use Python and function can be defined by lambda.

3. Performance
   3.1. "First Obama"

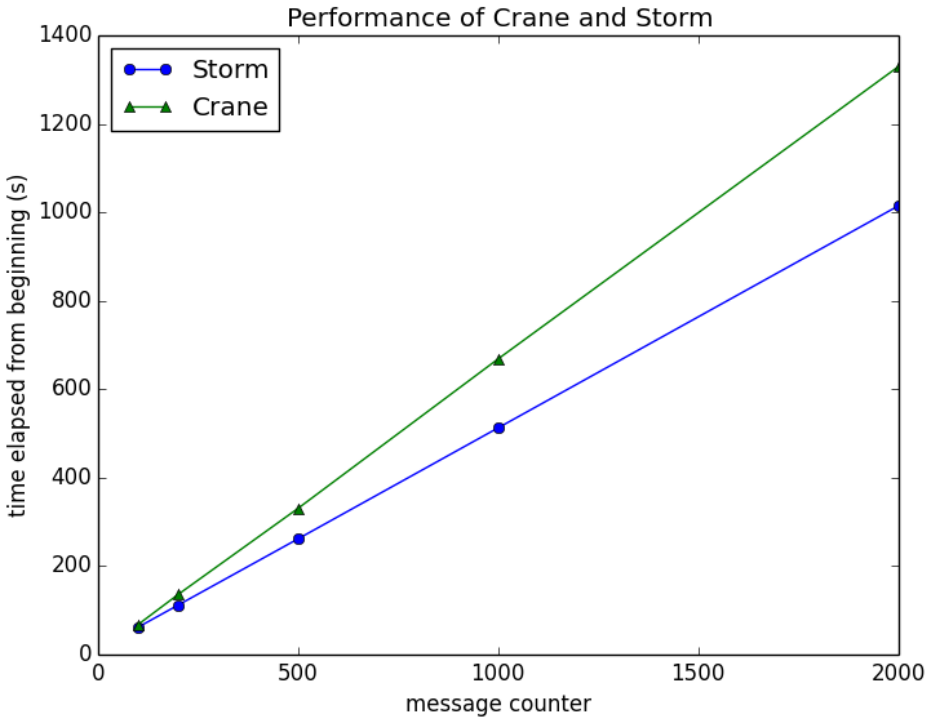
   This application is to count the tweets that mentions Obama.

   As you can imagine, we brainstorm a lot of scenarios (data types, applications, users, metrics) where we may beat Storm in some way. But we have to admit that Storm is hard to beat by our such limited resource. But if we have to name some performance advantages of our

lovely Crane, it is startup speed and hence the lower latency enjoyed by the beginning tuples. Still, this advantage is transient (sad but true). So for this comparison, we just use a small test dataset. Storm is much more complex than Crane, so its startup consumes more time. But its processing speed beats Crane with no doubt, so it catches up quite quickly.



3.2. “Average Word Length” & “Collect Header”



"Average Word Length" is to calculate the average number of words in tweets stream. "Collect Header" is to collect all the header information from each record.

In these two application, we compare the processing speed of Crane and Storm. The results of these two applications actually have negligible difference. This is a trivial result of our system design and implementation, which is not good enough. In detail, Crane's bolts are not parallelized for simplicity, the communication of tuples between bolts is UDP for simplicity, acknowledgement is via RPC for correctness ("exactly-once" semantics). If spout emits tuples more frequently than 2/s, the children bolts can't handle it quickly enough because RPC is costly. When timeout happens on the spout side, it resends the message but it will worsen the situation and **message explosion** occurs. So, we have to tune the spout to emit messages slowly due to the network bottleneck. And for fairness, we force Storm to align. Actually, if the input data frequency is strictly 2/s, these two lines should almost coincide. The slopes differ because we delay every message for 0.5s when it's its turn to emit, and the interval between messages includes other processing time additionally. The trend of their performance is quite trivial, reasonable, and predictable, so we just sample some points which are enough to represent the trend. And the disk I/O cost of "Collect Header" is minimal compared to the network delay margin, so we don't put a duplicate figure here.

3.3. Failure Recovery

We can't figure out how to calculate the failure recovery speed of Storm precisely, because it's long-running, sorry. But we do know that of Crane. It's about 6.2s, including the failure detection and restarting of a new machine.

3.4. Choice

Basically, please choose Storm for long-running applications. But if quick startup and easy configuration is desired, Crane is better.