1. Design:

In this MP we implement a distributed stream processing, called Crane, that has a high-level architecture and design similar to the Apache Storm. Our Crane system consists of four main parts, i) a core stream processing framework, ii) a client for user command line input that is capable of submit applications, monitoring application topology and intermediate streaming results, iii) a failure detection algorithm that is used to handle node failures during streaming and iv) a SDFS manager used to store result output files of our streaming applications.

Stream processing logic of Crane: The architecture of Crane is similar to that of Apache Storm and it consists of a master node and several worker nodes.

Nimbus: On the master node there's a daemon running called Nimbus (yes, we stole that name from Storm). Nimbus general acts as a coordinator. It holds a member list of all the other nodes including the hot stand by master node and all worker nodes to handle node failures. When user submit a job, Nimbus is in charge of distributing tasks to the worker nodes based on the topology specified by user. To simplify the implementation of Crane, we also assign spout to the master node, which reads the tuples from local file system and send to the next node in topology as a source of streams.

Supervisor: A supervisor daemon is running on each of the work nodes and waiting for instructions from Nimbus. Upon receiving the instruction from Nimbus, the supervisor will invoke the worker process (bolt) to receive streaming tuples the from upstream node of the topology and is capable of performing some simple operations to the stream including filtering, transformation and join, and finally send the processed tuples to the downstream node of the topology. In Crane, the intermediate results of each bolt process is saved into files and the final result output from the end of topology (the sink node) will be saved into the SDFS.

Failure Detection and Fault Tolerance: On Nimbus, ping messages are continuously sent to all the worker node every second and when the worker nodes receive the ping message will return an ack message to Nimbus. If ack is not received by Nimbus from a node, that node will be marked as offline. If the failed worker node is assigned as a bolt and has running tasks on it, Nimbus will detect that and reassign the job to another idle worker node, update the topology and restart the whole streaming process. Similar failure handling is used when a master node fails. An Nimbus daemon is also running on the hot standby master node when we start the system. The Nimbus on master node and hot standby master node both send ping messages to each other every second. If one of the running Nimbus detect the failure of the other, it will first check if the failed nodes is either the master node or the hot standby master node. If it is the hot standby master that failed nothing will happen, and if it is the current master node that failed, the hot standby master will change the status of itself into the master node and start to reassign the tasks and restart the whole streaming process. To distinguished the restarted process's streaming tuples from the previous failed one, in Crane, each streaming tuple is send to a bolt with a job ID corresponding to a process. If a worker node detects a different job ID of an incoming tuple, it will discard the previous results and start processing the new job.

User Input: To submit a job, use only need to provide a python yaml file. The yaml file has to contain the number of tasks required by the job and each start with a unique task id. And for each task user has to specify the following properties of that process: 1) Whether the task is the final task and require result file output, 2) the type of this task, either filter or transform, 3) the user specified function need to be processes and 4) the child task, where the processes data need to be sent to (the downstream node in job topology).

2. Applications:

To compare the performance of our Crane with Apache Spark Streaming, we have chosen the following three simple applications.

a). Transform positive reviews to negative reviews.

In this application we use a truncation of the Amazon Food reviews dataset and only keep the summary part of each reviews. First will filter the reviews to find out all the reviews containing "good" and then transform all the "good" into "bad" and output the final results.

b). Count number of product with average rating at least 4.0 and a least reviewed by 10 users.

This application uses the a truncation of Amazon product review dataset tat only contains the summary of the reviews (number of reviews, average ratings etc.). To run the application, first filter the raw input review files for the products that has a rating higher than or equal to 4.0. Then filter the filtered data again to find those products with at least 10 reviews.

c). Replace the 1 star ratings by 5 stars and count the overall 5 star ratings.

In this application we used a truncation of the Amazon food reviews and only keep the ratings, so it's a file containing only numbers. First transform the input to replace the 1.0 rating by 5.0 and then apply a filter to the transformed data to keep only 5.0 ratings and finally count the number of such entries.

3. Performance:

To compare the performance between Crane and Apache Spark, we measure the time used to complete the applications mentioned in section 2 with different input data sizes (counted by lines of input files). Since streaming is a continuous process, it is hard to time the execution time, especially for Spark where one have to manually shut down the job to collect the execution time. Therefore all the running times we measured are approximated values.

Performance without failure: For the best comparison, we used same setting and topology for all the three applications on both Crane and Spark. Five machines are used consists of 1 master node, 1 hot standby master node and 3 worker nodes.

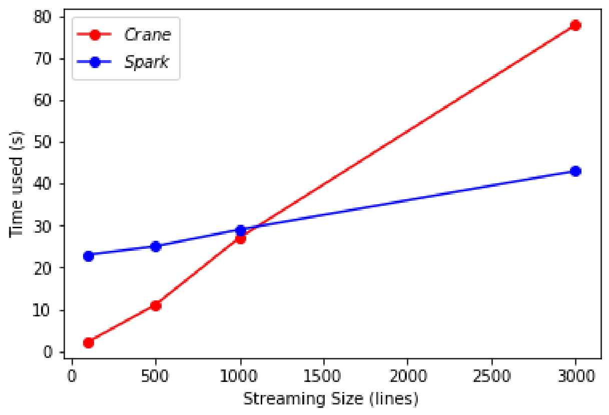

Figure 1: Application a): Transform positive reviews to negative reviews.

As shown in the plots, the performance of Crane is almost the same on the three chosen applications. This is expected since we observed that the speed limitation in Crane is not on the processing of tuples but on sending the tuples between machines using udp. As a result, the running times of all applications are proportional to the number of lines (tuples) of input and have almost the same slopes.

Spark did really good jobs on processing large amount of data and has a much better performance than Crane. But running with spark requires a larger start-up time, which is around 20 second in our VMs. On the other hand, our light weighted Crane only has a trivial start-up time that can be ignored. Hence Crane is better on dealing with small inputs. This is also expected since Spark is well optimized to achieve high performance on streaming data.

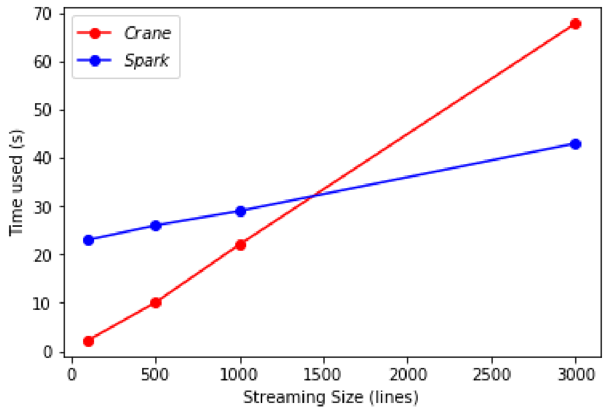

Figure 2: Application b): Count number of product with average rating at least 4.0 and a least reviewed by 10 users

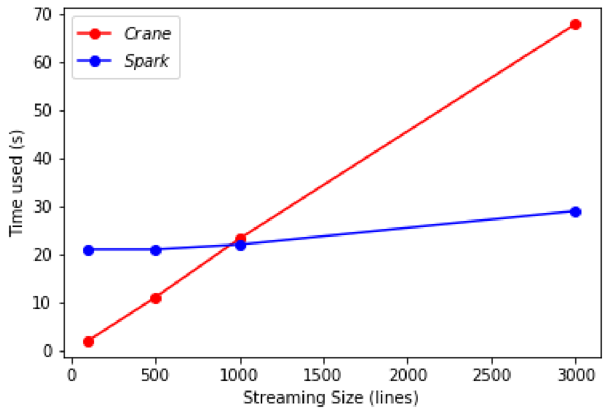

Figure 3: Application c): Count number of product with average rating at least 4.0 and a least reviewed by 10 users

Performance with failures: The performance of Spark with one node failure does not differ a lot from the measurement of performance without failure. This is probably because spark has a good fault-tolerance algorithm and support restart from checkpoints. For the Crane, is a worker node with running task fails, it takes about 2 seconds for the master to detect such failure and restart the job. Since in our design we will restart the entire job so in total the performance with failures for Crane is time used for aborted part of jobs + 2 seconds (failure detection) + performance without failure.