Machine Programming 4 – Distributed Stream Processing System

Yunsheng Wei (wei29) Neha Chaube (nchaub2)

Design:

In MP4, we built a distributed stream processing system called Crane. We will briefly describe some of the key designs in the following sections:

1. The system is composed of two components: master node called Nimbus, and slave nodes called supervisors. Nimbus is responsible for receiving jobs from clients, and assigning tasks to supervisors. Once being assigned tasks by Nimbus, supervisors will respond by starting separate threads for every task.
2. Nimbus maintains a map between supervisors and their assigned tasks. Once Nimbus detect a supervisor's failure, it will attempt to assign tasks on the failed supervisor to other live supervisors.
3. We use the Gossip group membership service in MP2 for failure detection, and use Observer pattern for the communication between group membership service and Nimbus.
4. In order to ensure the 'at least once' semantics for tuple, we adopt the ack mechanism like the one adopted in Apache Storm. We have a separate acker thread listening for ack message, and every time a tuple is sent from spout to bolt, or bolt to bolt, it will also send an ack message to acker. When the acker knows a tuple has been finished, it will notify spout.
5. The communication between Nimbus and supervisors, between clients and Nimbus are using Java RMI. The communication between bolts and bolts, bolts and ackers are using UDP. And in order to reduce packet loss, we allow clients to specify the rate for spout and bolts to emit tuples.
6. Our implementation supports parallelism for bolts. Also in our implementation, we assume Nimbus never dies, and it can support arbitrary failures of supervisors.
7. The whole project is implemented in Java, and we use Java's default serialization mechanism to serialize objects.

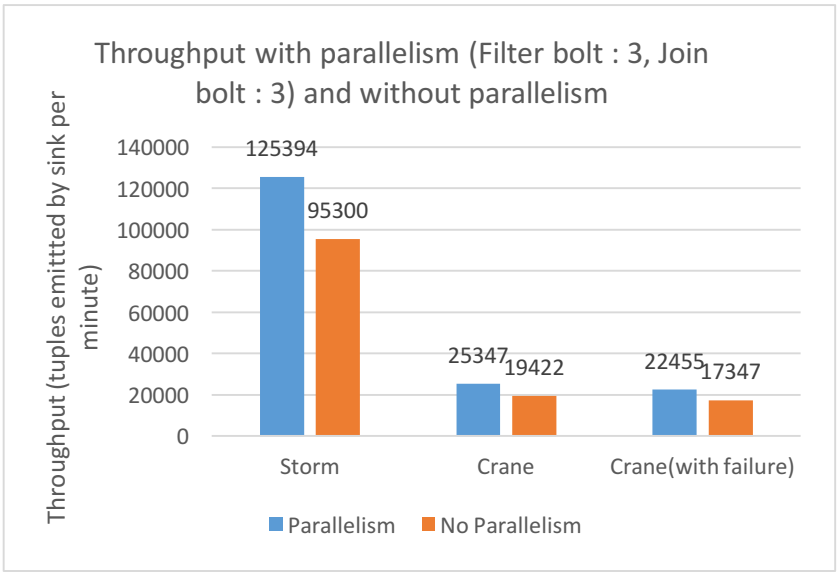
Applications:

We developed three applications for both Crane and Storm. The applications are based on the Tweet Trends dataset used in the paper Real-Time Classification of Twitter Trends.

All the following experiments are done on 3 VMs (1 Nimbus, and 2 Supervisors) for both Crane and Storm. We only do failure experiments for Crane (Fail one Supervisor). The metric we collect is throughput (Tuples emitted by Sink Bolt per minute). Because the dataset is not large enough, we manually make the dataset larger. We run both Crane and Storm for 5 minutes, and calculate the throughout.

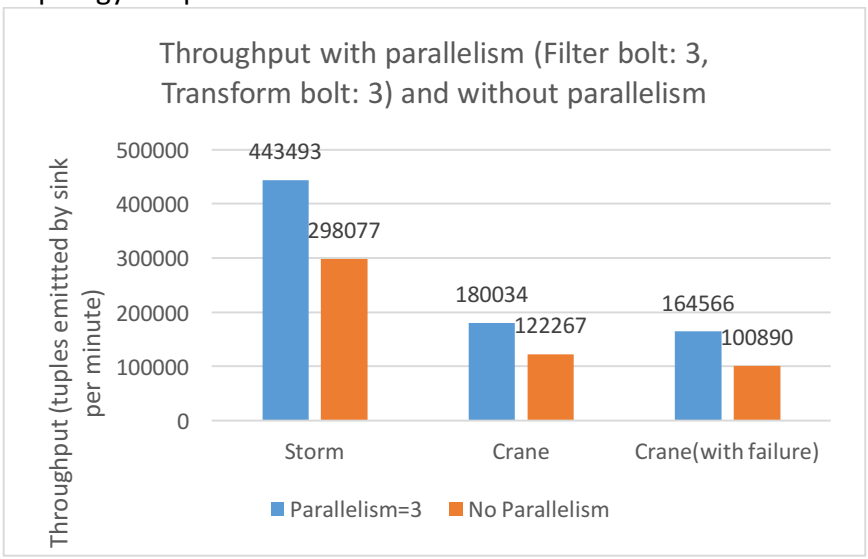
Our three applications are:

1. Find all users who tweet about a trending topic which belongs to the class "ongoing-event". The topology is: Spout -> Filter bolt -> Join bolt -> Sink bolt



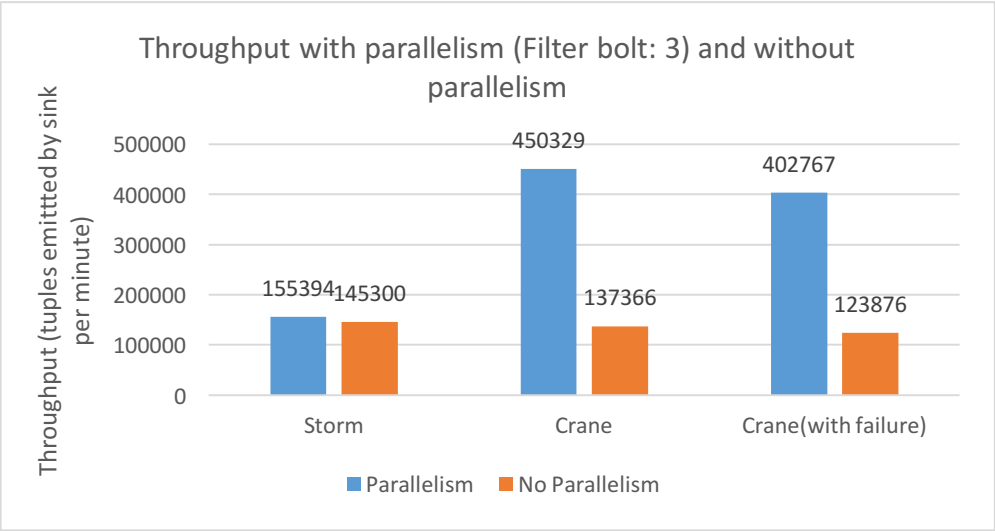
The topology is composed of a filter bolt and join bolt. For the join bolt, we need to join a tweetid with all user who tweet the tweetid, and on average one tuple will generate around 1000 children tuple. So in order to throttle the emitting rate, we throttle the spout emitting rate and also bolt emitting rate. So that's why Crane performs much worse than Storm. For failure case, the performance of Crane depends on the time for Nimbus to detect the failure and reassign tasks. Because the average detection time for our failure detector is 5 seconds, so the throughput does not decrease much.

2. Count the number of tweets for trending topic class "meme", "ongoing-event", and "news". The topology is: Spout -> Filter bolt -> Transform bolt -> Sink bolt



Because the the topology does not contain join bolt, so the rate for sending and receiving for every bolt is relatively balanced. So we do not throttle much on the sending rate. That's why the performance for Crane and Storm is relatively close. And the explanation for the failure cases is the same as the first application.

3. Filter all tweet ids for trending topic class “meme”, “ongoing-event”, and “news”. The topology is: Spout -> Filter bolt -> Sink bolt



The application is the easiest. For the parallelism = 3 case, we do not have any throttling, and the performance exceeds Storm.

Conclusion: One bottleneck for Crane is related to throttling. A better strategy may be to use TCP instead of UDP for sending tuples, because TCP has some back pressure mechanism. And another way to improve performance is to change the serialization method. We found Java’s default serialization is not so bandwidth efficient. Using Google Protocol Buffers may improve performance a lot.