CS425: Distributed Systems Fall 2015
Machine Problem 4
Report

Xuewei Zhang - xzhan160
Luis Remis - remis2

Design and Implementation

1. Membership Control, Leader Election, and File System

Our membership control protocol adopts the concept of SWIM protocol. If no machine fails, the message load is small (always O(1)). Since infection-style dissemination will increase the message load when no one fails, we use TTL broadcast to do dissemination. We use our membership protocol as a base for our MP4 implementation. We implemented the bully algorithm for the leader election, which proved to be very fast. The distributed file system in MP3 is used as a robust data carrier in crane

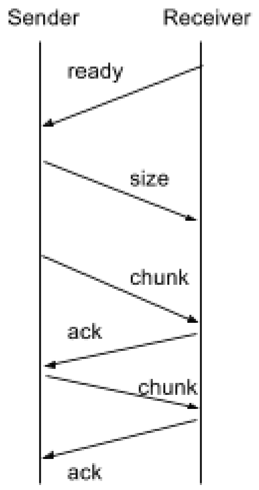
2. Master and Supervisor in Crane

For the implementation of our Crane System, we design and architecture using a master running in a single machine and a Supervisor running on any other machine that will be handling tasks from bolts (the machine that has the master running can also have a supervisor running on it to handle tasks). This way, the master has knowledge of all the supervisors and the tasks that they are handling. The topology is created and defined in the master machine, and the master communicate with the supervisor in all the other machines to create tasks. Since a bolt can have several tasks, the tasks are round-robin splitted in other to achieve load balance.

The Supervisor is in charge of creating a new task in a machine, and assign a port where the task will be listening. Once all the tasks are created with an assign port, the bolt subscription process starts, using the assign port to agree in a new port for a peer to peer TCP connection. This way, task of each bolt is connected using a dedicated TCP connection to any other tasks that are subscribed. On the supervisor, the new bolts created are single, one-threaded tasks that will receive tuples, process them and emit a result in case it is necessary.

The master has all the information about all the bolts and tasks around the system and splitted in the different machines. This way, and by using the membership system implemented in MP2, when a failure occurs in a machine, the master reassigns all the tasks that were being processed in that machine to any other healthy node.

3. Communication between Bolts



The main bottleneck of stream processing is the communication from Spout to Bolt or from Bolt to Bolt. We create one TCP channel, one message queue and a pair of socket communicating thread for each pair of communicating tasks' thread. In the application, when task emit() a tuple, it will be added to the message queue according to its destination. (By default we support shuffle grouping or all grouping.)

The pair of socket communicating thread is always talking to each other. We have a "hand-shake" protocol for the data exchange. Receiver thread will first claim him available for receiving data (to prevent TCP buffer overflow and provide back-pressure). Then Sender thread will pack all the tuples into one big buffer, and send the buffer size to Receiver. If the buffer is too big, it will be divided into smaller chunks to send. And for each chunk, Receiver will send back ack message. The protocol is proved to be very fast.

4. Fault Tolerance

The fault tolerance in this program contains two steps:

a. Make sure threads does not crash for broken pipe. When senses other machine's failure, each Bolt's task thread do a cleanup and finish. This can be done by installing a signal handler for SIGPIPE, and a flag for task thread to finish(does not finish supervisor thread). When master node gather the information about alive machines in the cluster, master will restart every task.

b. Master keep a table of sent tuples and acked tuples. This is done by a std::Map(tuple, vector<bool>), and each tuple contains one tuple ID(unique integer). Basically, when Spout send a tuple, one entry will be stored at the table in master node. When a tuple get filtered out or processed by the last Bolt, Bolt send a ack() message to master. When master receives an ack() message, it will mark one or more of the <bool> as true, depends on the topology and the ack() emitting point. When all the <bool> is true, it delete the entry. When master restart the topology, spout will resend every tuple in the Map. The hard thing is, topology can have many exit points. And after failure, some exit point ack() the tuple but others not. In that case, we add certain fields to the tuple, to indicates which branch it should go. And the Bolt need to re-implement the emit() function.

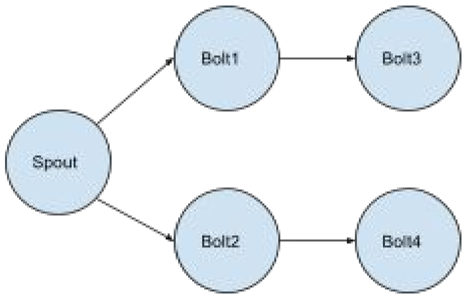
5. Implementation Details

We implement our system using C++. All bolts inherits from a Bolt base class, which handle the subscriptions, the thread for receiving and enqueueing tuples, and other basic functionality. Any new bolt needs to inherit from this class, and implement the run() method that will define its behavior, internal state, or any other specification. Because of this, it is trivial to implement a transformation, a join with static or dynamic database, or a filtering.

Also, a bolt can implement the method emit() if it want to use some emission of tuples that is more sophisticated that shuffle grouping or all grouping, which are implemented in the base class. By implementing this method, the emission of the tuples can be done in a shuffle manner or using some form of identification for sending to a particular task. These more sophisticated are implemented by the use of a hashing function.

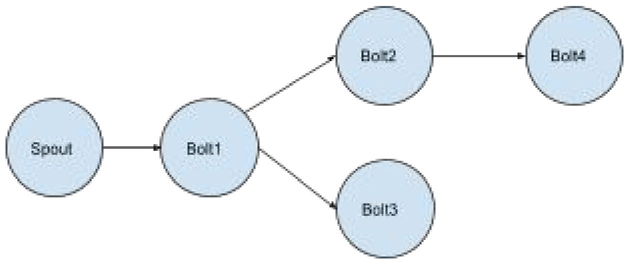
The spouts are implemented in a similar way. A base class Spout will have all the common methods such as emitting the enqueued tuples and communication threads, and new spouts needs to implement its own tuple generation, based on the origin of the data (random, files, other systems using TCP connections).

Apps for Crane



We designed 3 apps as benchmark applications of our crane system.
The first App involves the processing the trace of thirty days' worth of all wide-area TCP connections between the Lawrence Berkeley Laboratory (LBL) and the rest of the world. This App aims to count the number of TCP connections per protocol and the number of bytes sent and received per protocol, as well as output the most used protocol in terms of connections open and bytes transmitted.The spout gets the protocol from the file, the bytes sent and the bytes received, so the tuple will have 3 elements. Bolt1 will have a hash table for each protocol and count the number of connections. Bolt2 will have a std::Map for each protocol counting the number of bytes for each protocol. Bolt2's emit() is a special case. Each time the std::Map filled with 1000 entries, we will send a big tuple containing all the pair in Map, and clear the Map. This is for efficiency issue. Bolt 3 will receive from bolt1, and will display the protocol with more connections. Bolt4 will receive from bolt2 and will display the protocol with more connections.

The second App was designed to process a trace that contains approximately one year's worth of all HTTP requests to the University of Calgary's Department of Computer Science WWW server located at Calgary, Alberta, Canada. This app aims to rank the most visited websites and count the most downloaded jpeg files. We then first filter all gif files which are of no interest.



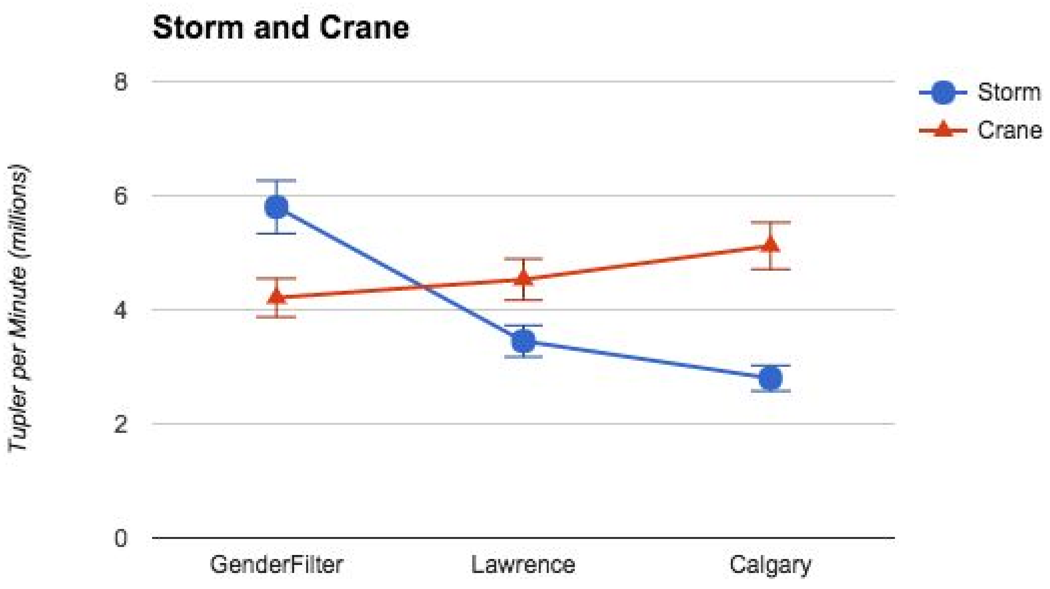
The spout will take the file after the GET in every line.
The first bolt will filter all .gif files and send the tuples to bolt2 and bolt3. Bolt 2 will have a count of every html file that appeared (using a std::Map to store the count of each html). Bolt2 will send a tuple "page, count" to bolt4. The emit() is also specialized like the first App. Bolt3 will receive from bolt1 and count all jpeg or jpg files. Finally, bolt4 will receive from bolt2, and update its local std::Map about page-visit pair. When see a new "most visited" page, print it.

Finally, the last App was aimed to do speed testing and comparison with Storm. For this, we use a file containing tweets to generating fictitious gender of the person tweeting, age and the twit id (unique number). We then implement a Spout generating the tuples, a bolt that will receive from this spout and filter the tweets with gender "unknown", adding an extra string to the tuple based on the gender, and send this tuples to two other bolts. One bolt will be in charge of counting the number of tweets emitted by males and the other bolt counting tweets emitted by females.
In this case we have a bolt for filtering and transforming, and other bolts that will have internal state for keeping global counters.

Testing Performance

For testing the performance of our system, we run each of the above mentioned applications in both system using 7 machines and a max of 4 tasks per bolt, for the first two applications, and different values for the last application until Storm started to perform better, so that we can analyze why that is the case.
For these experiments, we use machine 1 to serve as an administrative machine, which would only be in charge of generating the topology and read file to feed the Spout. We did 8 runs on every test to get accurate data.
Since Storm by the use of Zookeeper also uses a node only for administrative purpose, we choose also this configuration in order to have a fair comparison.

For all the tests, we output the result into the distributed file system, later joining them in the server, so that printing results did not took execution time. In the demo we use standard output to show the results as it was only necessary to prove that the system was working faster. We run several iterations over the input files (one hundred pass over the files) and then measure the time it took to process all the tuples



We tuned the last application until we reach the point where Storm was working faster under the same conditions (topology, i.e., number of bolts and tasks per bolt). In order to achieve this point, we thought about possible causes for slowdown in our crane implementation. At the moment, the implementation handles all the tasks on different bolt inside the same process under the supervisor in the machine. This means that we have several threads belonging to the same process, that are competing for the resource assigned to that process. We realized that if we increase the number of task per bolt to a number higher than 25, our supervisor would not experience good scalability, and we manage to make Storm run faster in this case. This issue in our system can be solved by managing tasks or group of task as different processes in the setup, and making a better division of the administration and task works, which is part of our future work.

Xuewei Zhang - xzhan160
Luis Remis - remis2

	TestCase	StormTime	StormBar	CraneTime	CraneBar
1.	Gender	4min10s	26s	3min 30s	13s
2.	Law	5min 44s	50s	6min 20s	60s
3.	Cal	6min 10s	35s	4min10s	30s

References:
Lawrence Berkeley Laboratory : http://ita.ee.lbl.gov/html/contrib/LBL-CONN-7.html
University of Calgary: http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html