CS 6381 Final Report – Google Cloud Pub Sub Team:

1. Overview
2. Experiment Criteria and Design
   1. We created a set of applications to test the message sending/receiving efficiency of the Google Cloud Pub Sub service based on our overall understanding of the experiment designers’ guidelines and the time available. We established pub sub topics, project and topic ID’s, and a JSON key with the help of the Google Cloud Pub Sub user interface online. On Ubuntu 18.04, we set up the Google Cloud SDK and configured information based on the project and topic ID’s and the JSON key. With this, we were able to run our sets of applications.
      1. Data flows between our applications, pub.py, broker.py, and sub.py.
         1. Pub.py sends messages to broker.py via ZMQ continuously. This involves establishing a ZMQ context, getting the IP address via a socket command, and binding the socket to a TCP path that includes the IP address and a given port number. Then a “while True” loop, we loop through the files in the messages folder, get the text from there, append it to the message size, which is how many characters there are in that text, current timestamp, and iteration count, and send off the combined message through the socket.
            1. The message files in the messages folder contain text that vary in lengths and are created with the help of messageGenerator.py, which generates a string of random characters to a specified length and saves it to the messages folder as a message file.
         2. Broker.py receives messages from pub.py via ZMQ and sends messages to sub.py via Google Cloud Pub Sub continuously. This involves passing the project and topic ID’s to a class. In this class, a given IP address and port number are used to connect to the socket and a filter is established based on the topic keyword. In a “while True” loop, a call on the poll() function occurs and when a topic match is found, incoming messages are received, a timestamp is taken, and this information is passed to the measureTime() function that parses the message, finds the time difference between the receipt time and the time the message was sent from pub.py, and assembles a new one based on the message size, current timestamp, this calculated publisher to broker time, success rate, and message content. This new message is converted into bytes and then sent out via Google Cloud Pub Sub.
         3. Sub.py receives messages from broker.py via Google Cloud Pub Sub and saves the messages into CSV files when specified numbers of messages have been received. This involves listening for incoming messages via Google Cloud Pub Sub. When the message is received, a timestamp is taken and both are sent to the extractMessage() function. This function parses the message, calculates the time difference between the receipt time and the time the message was sent from broker.py, appends the message ID, publisher to broker time, and this calculated broker to subscriber time to a message list, and returns the message size and publisher to broker success rate in the form of a dictionary. When the size of the message list equals the given message limit, the messages from the message list, containing the time measurements, are saved into a time measurements CSV file that includes in its name the message limit, message size, publisher to broker success rate, and UUID for clear distinction.
         4. Messages contain message ID’s and times it took to get from pub.py to broker.py and from broker.py to sub.py.
         5. We ran tests using shell files to examine the effects of varying producer/consumer conditions based on the following:
            1. We treated broker.py as the producer since it sent messages via Google Cloud Pub Sub and sub.py as the consumer since it received messages via Google Cloud Pub Sub.
            2. Within a test shell file, we fetched the JSON information and configuration values for Google Cloud Pub Sub in each command that involved running a producer or consumer or pub.py.
            3. The order of running from 1st to last is as follows: sub.py, broker.py, and pub.py.
            4. Each test included variation in message length based on the messages in the messages folder.

The message files vary in terms of total number of characters as follows: 22, 149, 6815, 155355, and 761135.

* + - * 1. There was some sleep time applied to wait for processes to finish before attempting to delete the processes for broker.py, then pub.py, and finally sub.py and running clear.py to clear future messages in a way since we have observed that it seems like Google Cloud Pub Sub sends old messages that the recipient has not received the next time it connects for receiving messages. The other way we found to essentially empty out the old messages in a way was to start up sub.py and check to see that all the old messages have been received as logs stop showing up in the output.
        2. Run the following tests:

|  |  |  |
| --- | --- | --- |
| Test | Definition | Times ran |
| 1 producer vs. 1 consumer | 1 broker.py, 1 sub.py | 1 |
| 1 producer vs. 5 consumers | 1 broker.py, 5 sub.py | 1 |
| 5 producers vs. 1 consumer | 5 broker.py, 1 sub.py | 1 |
| 8 producers vs. 8 consumers | 8 broker.py, 8 sub.py | 3 |
| 25 producers vs. 25 consumers | 25 broker.py, 25 sub.py | 4 |

* 1. We used calculateStatistics.py to read the data from the time measurements CSV files and save a CSV filled with statistics and calculateStatisticsAll.py for similar purposes with the additional feature of generating graphs based on the calculations.
     1. Here, we set a for-loop to check for CSV files in a folder. When there’s a match, we call readCSV(), which takes in the file and opens it, iterates through it, and appends the write and read times to lists as well as write and read success rates, which are based on how many time rows there are divided by the total iterations for a file. For latency and throughput, using this collected information, we get the minimum and maximum values for times by using the min() and max() functions respectively, average by summing the tuple-converted data list of float-converted times and dividing that by the length of the data list, and standard deviation by getting the variance for the result of dividing the sum of the data list by the length of the data list and setting it to the power of 0.5. these findings are added to a calculations CSV file with the current timestamp and UUID in the title to distinguish the files apart.
     2. calculateStatisticsAll.py featured generating graphs by passing in the file name that is parsed from the path name. Using matplotlib and pandas, graph settings are configured with the X-axis showing the various test conditions and the Y-axis showing the values for the parameter of interest as graphs are generated.
  2. We used hardwareMeasure.py to continuously see the latest hardware statistics, including CPU, RAM, and storage. In a “while True” loop, we printed the functions that return a dictionary including the max average values for each of these parameters. In each function, we appended the parameter fetching using psutil to its own list and from that list, we find the maximum value using the max() function and average value by summing up the tuple-converted data list and dividing that by the length of the list.

1. Experiment Results
   1. The following includes our observations of trends among the graphs.
      1. In terms of standard deviation, average, max, and min for publisher to broker success rates and broker to subscriber success rates, the 1producer1consumer test yields low results, the 1producer5consumer test yields moderate results, and the other conditions as well as when all data is considered yield the highest results. For broker to subscriber success times, this trend appears similar for standard deviation, average, and max, but for min, 1producer5consumer, 25producer25consumer, and 5producer1consumer tests yield moderate results as so do publisher to broker times in terms of standard deviation, average, and max. For publisher to broker times min, there is less variation between the different conditions with slightly less results for the 8producer8consumer tests and when all data is considered. One thing though is that the 25producer25consumer test did not seem to work to full efficiency on the computer we tested on as the computer froze and we only have 20 results for 1 run of that test.
         1. From these observations, it seems that Google Cloud Pub Sub throughput, which corresponds to the broker to subscriber success rate, appears mostly high when there are more producers and especially consumers. When we increase the number of producers and especially consumers, the latency increases. Perhaps this is due to increased competition for access to Google Cloud Pub Sub, which may create a bottleneck effect. A similar pattern occurs with ZMQ, but the results are more moderate for some conditions. When there are too many producers/consumers or more producers than consumers, this yields moderate results at least in our case. For the most part though, it seems Google Cloud Pub Sub can send messages efficiently in certain aspects like ZMQ.
      2. For the base file, it appears the hard disk memory stays roughly stable while the readIteration.py test is run. RAM and CPU increase from the 1st to the 2nd load.
         1. The computer we used had other activities going on outside of the VM, so perhaps this contributed to the RAM and CPU fluctuations. The computer has a lot of hard disk memory, so changes to the memory should appear minimal as long as large-sized items are not added or removed.
2. Conclusions
3. References