PERFORMANCE REPORT (TEAM: CARD_READERS)

Note : This report contains multiple pages. Please download the pdf file to see the full report.

1. **Overview of the analysis**
   To evaluate the performance of our FaaS (Function-as-a-Service) system, we conducted a weak scaling study to compare the relative performance of the local, push, and pull execution modes. The local implementation (using a single Python multiprocessing pool) was treated as the baseline for comparison, as it leverages a highly optimized library without introducing additional mechanisms or overhead introduced by our project.

   The goal of the weak scaling study is to assess how well the system maintains consistent performance as both the workload and the resources scale proportionally. In other words, we aim to determine whether the system can handle larger workloads efficiently without significant degradation in performance.

2. **Weak Scaling Study Design**
   We selected a scaling factor of 5, meaning that for every additional worker, the number of tasks increases by 5. The workload and resources were scaled as follows:

   1 worker = 5 tasks
   2 workers = 10 tasks
   ...
   16 workers = 80 tasks

   The study was conducted across five configurations: 1, 2, 4, 8, and 16 workers (corresponding to 5, 10, 20, 40, and 80 tasks). This allowed us to observe how the system's performance evolves with increasing load and identify any performance bottlenecks.

   Each pull/push worker was spawned with a single process running on it. 1 worker = 1 process. So to simulate 16 workers, 16 individual terminal were spawned. Note that this would not be the same as creating one worker with 16 processes. This was done to keep the focus of the study on what happens as you add independent workers/resources to a system. Increasing the number of processes on a single push/pull worker and linearly increasing the number of tasks did not seem like it would really answer the purpose of the study, which, reiterating, is to understand what happens with increasing load while increasing system resources. Increasing

the number of processes on the same worker can be done but since that is being taken care of by Python's Concurrent.Futures library, it would not tell us much about our implementation of the dispatcher.

3. **Task Types and Their Purpose**

   To gain a comprehensive understanding of the system's performance, we evaluated two types of tasks:

   No-Operation (No-op) Tasks:

   - This task simply prints a message: *"Task: No-op completed."*
   - Purpose: To measure the system's inherent overhead (e.g., communication, coordination, and synchronization) in isolation, without introducing additional computational load. By isolating task latency from system-induced overhead, we can better understand the design and implementation costs of the system.

   Sleep Tasks:

   - This task involves sleeping for 1 second before returning.
   - Purpose: To simulate the system's behavior when processing tasks that require actual computation or processing time. This helps evaluate how the system performs under workloads that include both system overhead and task execution time.

4. **Running the performance tests**
   a. To find the client script cd app/performance and you will find in that directory a test_client.py. We set performance tests to run with pytest library so ensure you do not change the name of this file, nor the names of the functions within it.
   b. We wrote a script app/performance/test_client.py that contains the code for the performance tests. We set up a conftest.py and used pytest fixtures to allow user to provide the configurable parameters
      i. --workers=<int>
      The number of workers you have set up to test performance with. Note that this needs to be provided by the user in order for the test_client.py to determine the number of tasks it needs to perform (scaled as 5 times the number of workers you provide). This does not spawn the worker processes so please ensure your spawned number of processes are entered and set up correctly.

ii. --optype=<noop or sleep>
Whether you want test_client.py to run no operation (noop) or sleep(tasks). Use noop to understand system overheads and sleep to get insights on latency and throughput of the system. We did also set up a CPU intensive computation called "heavy" which can still be run, but we found that it did not provide useful insights compared to sleep so we have chosen to only analyse with noop and sleep tasks.

iii. --mode=<local, push or pull>
This does not directly influence how the performance tests run but gets used to print information in the console as well as write to a performance_test.csv file which can later be used to visualize the data.

c. To run the performance test
$ cd app/performance
$ pytest -s test_client.py --workers=<num of workers> --optype=<noop or sleep> --mode=<local, push or pull>

-- for weak scaling

-- for strong scaling

d. Ensure you have set up the correct mode of the dispatcher as well as the correct number of workers before calling this performance test.Each worker has to be set up on a separate terminal. Ensure dispatcher is spawned first and then the workers so dispatcher can be ready to register workers. Then run the performance test.
e. The performance test waits for the functions specified to run and records the total time it takes for 5*number of workers tasks to finish. This data about number of tasks(load), number of workers and total time taken is written to a csv called performance_test.csv. We ran each test multiple times and chose to retain the best time. Data can be found here: Project Card Readers
f. We choose to use the best time and not the average time of various runs to account for any latencies introduced by other background processes on our laptop.

5. **Key performance metrics**
   a. **System overhead:** System overheads refer to the extra time and resources consumed by the system for operations such as communication, coordination, task scheduling, and synchronization, beyond the actual task

execution.

Formula 5.1:
Overhead time = Latency of push/pull - Latency of ideal(local)

b. **Latency:** Latency is the avg time taken to complete a single task, including both system overheads and task execution time. It provides insight into the responsiveness of the system for individual tasks. High latency can indicate bottlenecks in communication or processing.

Formula 5.2:
Latency = (Total Time taken for n sleep tasks) / (n i.e number of tasks)

c. **Throughput:** Throughput is the total number of tasks completed per unit of time. It measures how efficiently the system handles a workload as a whole.

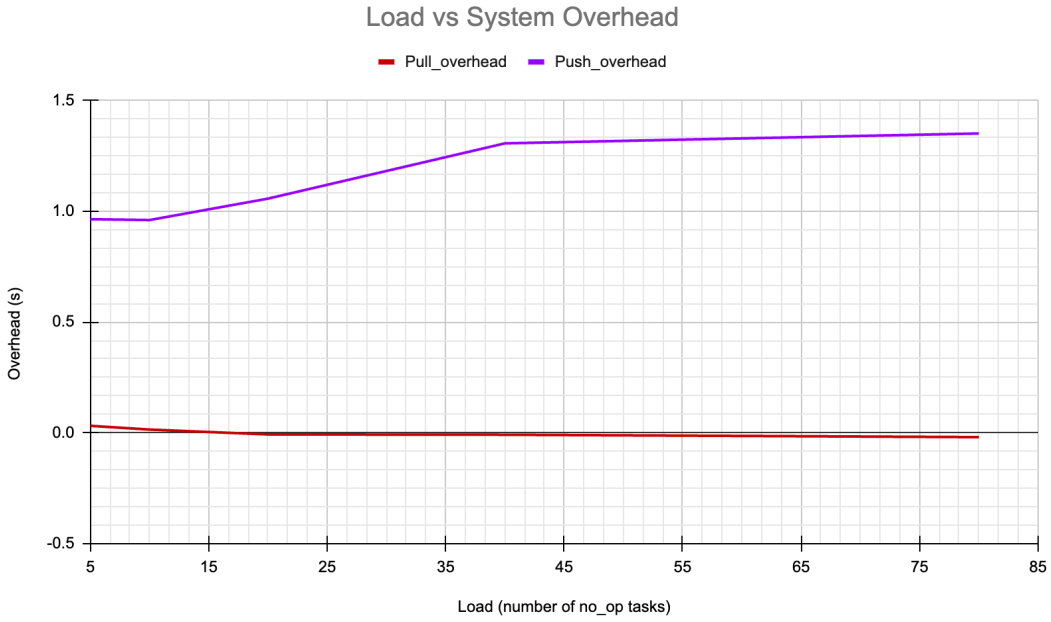
Formula 5.3:
Throughput = (n i.e number of tasks) / (Total Time taken for n sleep tasks)

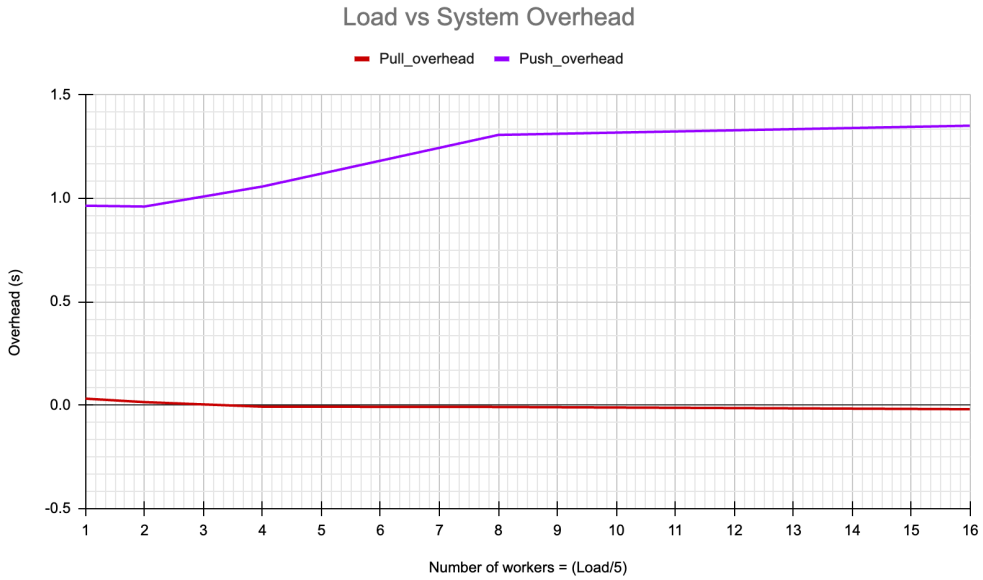
d. **Efficiency**: Efficiency measures how effectively the system utilizes its available resources as it scales. It compares the actual latency to the theoretical maximum latency (ideal scaling shown by the local). Efficiency helps identify diminishing returns as more resources are added.

Formula 5.4:
Efficiency = [(Actual Latency of Pull/Push mode) / (Ideal Latency of Local mode)] * 100

6. Analysis of the graphs

a. Load vs System Overheads





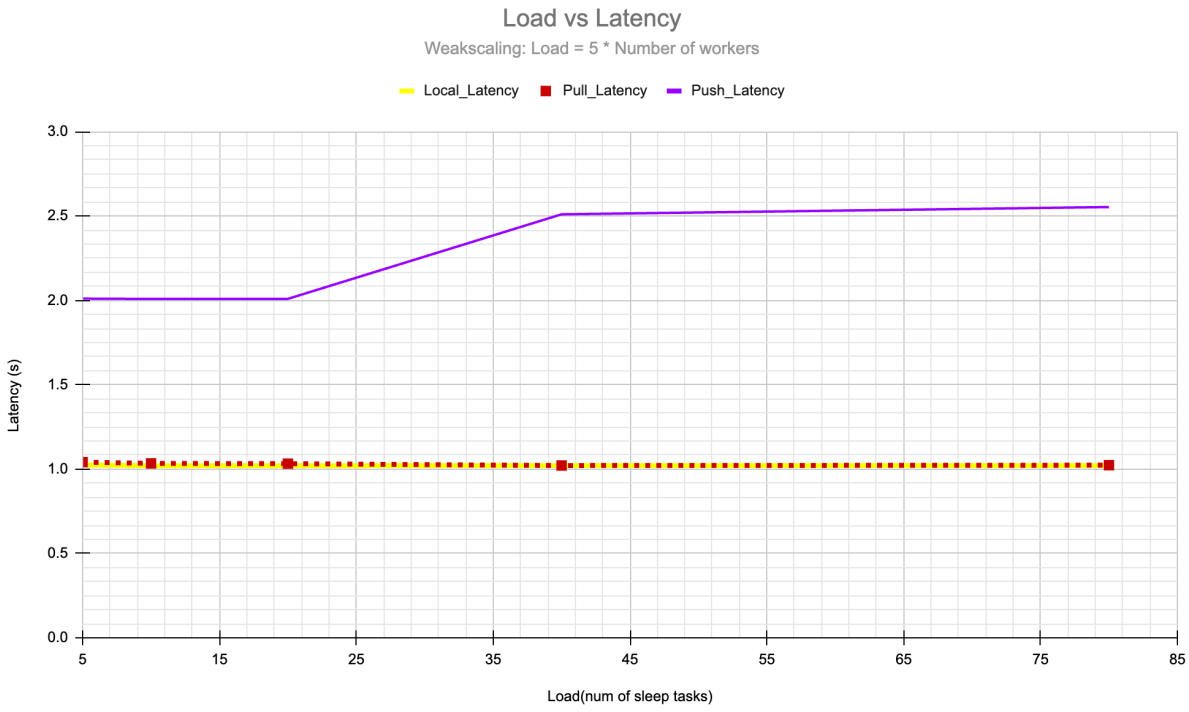
Process: To compute the system overheads, Formula 5.1 was used. The latencies were first computed using Formula 5.2.

Observations: The pull mode showed close to zero overhead relative to the local mode's performance, and even showed slightly negative overhead after 4 workers (20 tasks). The push mode was showcasing a linearly increasing overhead that then stabilized after 8 workers (40 tasks).

Interpretation: The pull mode is showcasing zero and negative overheads because there is minimal contention for resources when communicating, and this gets better than local upon adding workers because the local Python Multiprocessing Pool probably faces communication and synchronization overhead as all the worker processes have to coordinate on a single pool. On the other hand, the Pull Workers don't compete for resources amongst each other as they are running on independent terminals and not being coordinated by a single worker like in the Local Mode.

NOTE MOVING FORWARD: The number of workers will always be load/5. So I will only present graphs with Load on the x axis.

b. **Load vs Latency**



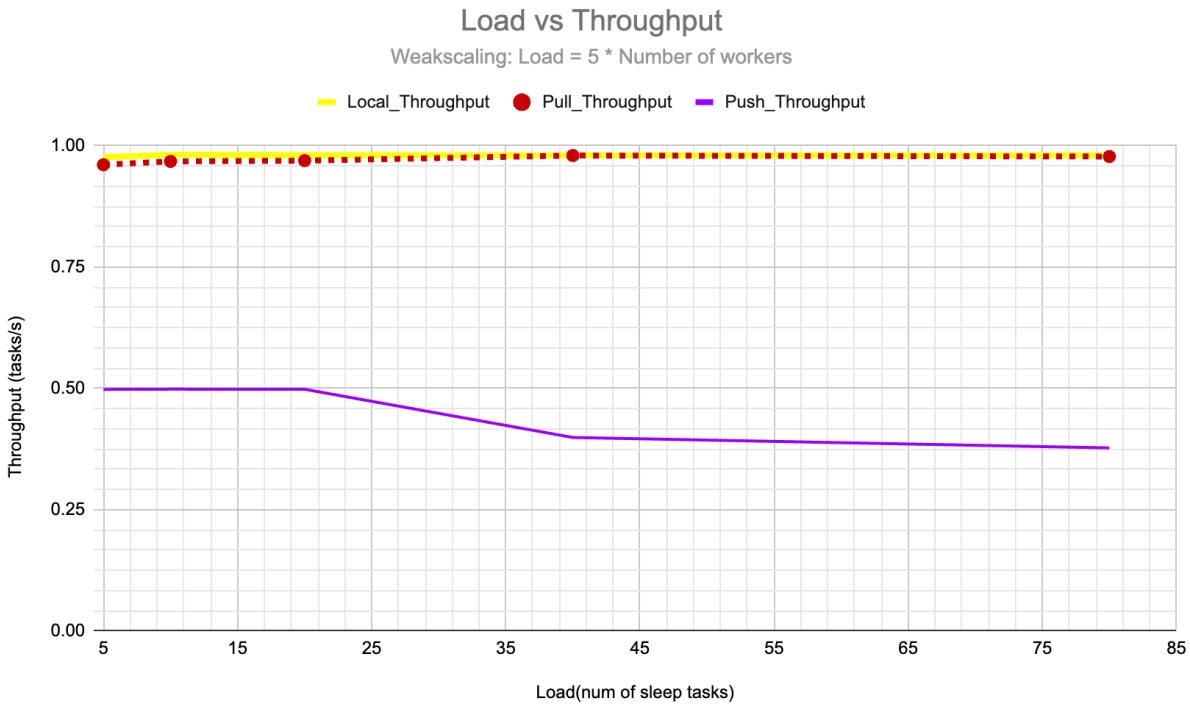
Process: To compute the latency, Formula 5.2 was used. Time taken for each task was recorded and then an average was taken to represent time needed per task. Tasks

were of sleep(1s) type.

Observations: The pull mode's latency stayed constant and aligned with the latency of ideal local dispatcher. The push mode was showcasing a linearly increasing latency that then stabilized after 8 workers (40 tasks).

Interpretation: The fact that latency for both pull and push stay relatively constant with increasing load aligns with the system expectations. It makes sense that the latency is constant at around 1 second as the sleep task was to simulate a workload of 1 second. The pull mode being close to ideal performance indicates there is minimal communication overhead which is an excellent sign. On the other hand, push consistently took almost twice as long! There seems to be overhead or some bottlenecks in push. There was a system of overhead of 1 second in the earlier graph that seems to be contributing to the 2 second latency for push.

c. Load vs Throughput

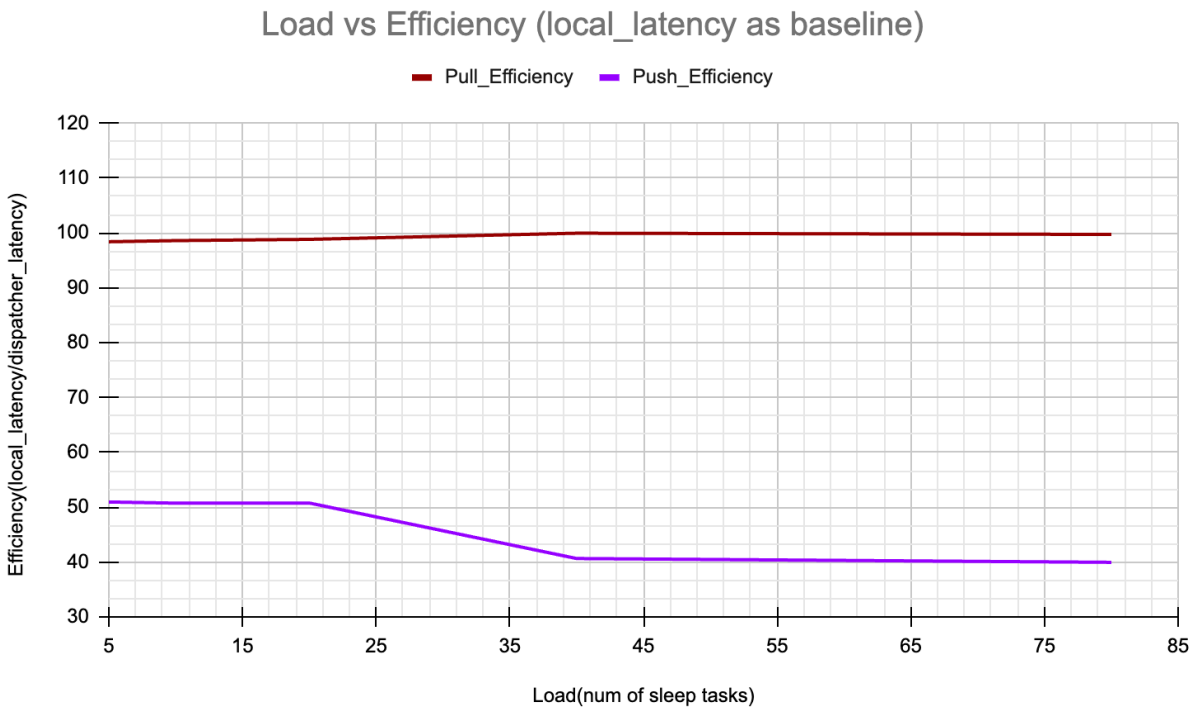


Process: To compute the throughput, Formula 5.3 was used. Time taken a total of n tasks was computed and plugged into the formula.

Observations: The pull mode's throughput stayed constant and showed a slightly linear increase. This aligned with the throughput of ideal local dispatcher. The push mode was showcasing a constant and then linearly decreasing latency that then stabilized after 8 workers (40 tasks).

Interpretation: The fact that throughput for both pull and push stay relatively constant with increasing load aligns with the system expectations. The pull mode being close to ideal performance indicates there is minimal communication overhead which is an excellent sign. On the other hand, push consistently was only able to handle half a task per second! There seems to be overhead or some bottlenecks in push.

d. Load vs Efficiency



Process: To compute the efficiency, Formula 5.4 was used. The computation was based on latency and the ideal local worker was the baseline.

Observations: The pull mode's efficiency was nearly 100% and that of the push was only 50% of the local dispatching mode.

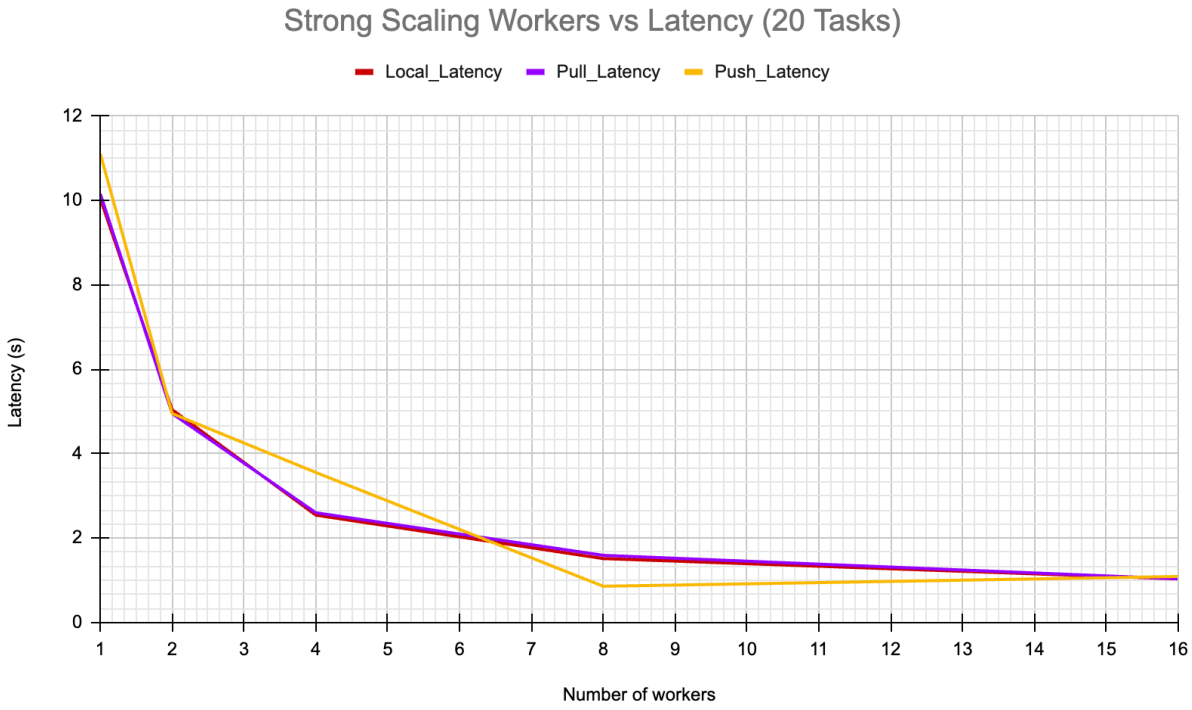
Interpretation: The fact that efficiency for both pull and push stays relatively constant with increasing load aligns with the system expectations. The pull mode being close to ideal performance indicates there is minimal communication overhead which is an excellent sign. On the other hand, push mode consistently was only half as efficient! There seems to be overhead or some bottlenecks in push.

e. Strong Scaling Worker vs Latency

Process: Measure the time take to complete 20 no_optasks. Vary number of workers in each mode.

Observations: Lower latency as we increase workers.

Interpretation: All modes are scaling proportionally to the task. Graph reaches plateau at the 16 workers indicating that the parallel processing is maximized and the system is hitting the limit of the sequential part of the code.



7. **Conclusion: Opportunities for improving the system**

   We found the overhead, latency, throughput and efficiency of the push mode deeply curious and we dove deep to discover the bottlenecks.

   At the heart of the problem was… THE HEARTBEATS!

   The push dispatcher is set up to have two threads-one to handle the main message processing and another to check the heartbeats being sent by the workers and determine if they need to be marked as dead. Having two threads that both run continuously with busy while loops leads to too many context switches which was bogging the system down. They also both updated a dictionary with information on worker ids, active tasks, and heartbeats and this required acquiring a global lock. This was causing sequential bottlenecks in the dispatcher.

   We experimented by increasing the HEARTBEAT_INTERVAL from 1 second to 10, 30, 60. At 60 seconds, the dispatcher was freed from handling too many heartbeats and was able to focus on dispatching tasks and updating their status, results on Redis. The issue with having too large an interval between sending heartbeats was that the

dispatcher would find out too late if a worker was dead. By then it might have already dispatched a bunch of tasks to a dead worker. As we were not taking care of requeueing these tasks, it would lead to a lot of tasks marked as FAILED due to WorkerFailureExceptions whenever the dispatcher did eventually detect that a worker had died/missed a heartbeat.

On the side of the push worker too, there were two threads-for the main task processing, and the second for periodically sending the heartbeats. As they were both using a single socket, they needed to acquire a lock before sending any task related messages as well as when sending heartbeats. This also introduced sequential bottlenecks in a system that had to acquire locks to perform its most frequent operations!

The performance analysis brought this to light and highlighted areas of improvement which are as follows:

- **a.** Make both push worker and push dispatcher work with a single thread to avoid the usage of global locks in the system.
- **b.** On the dispatcher, perform checks on the heartbeats only when no other task related messages have been received from the workers.
- **c.** On the pull, do away with the second thread and simply send a heartbeat in the same while loop that is listening to messages from the task dispatcher.

This was eye-opening and made the value of performance analyses abundantly clear to us.