Group_09_MP4

Daniel Petrisko (petrisk2) Xuefeng Zhu (xzhu15)
Yanwen Li (li206) Abhishek Sharma (sharma51)

Part 1: Implementation & Design Decisions

Given below are the steps we followed in order to implement Dynamic Load Balancer.

Step 1: Bootstrap Phase

g09 (172.22.156.12) node is chosen as master node, and g09-s (172.22.156.72) as slave node. In bootstrap phase, the local node will transfer half of the workload to the remote node.

Step 2: Processing Phase

a) After initialization step is finished, several threads are initialized: worker_thread, hardware_monitor, state_manager, transfer_manager, adaptor.

b) Worker Thread:

The worker thread executes the computation job (increment each element of vector) on a separate thread. Worker thread keeps working until both nodes finish all the jobs in their queues.

While processing each job, the worker thread will limit its utilization to the throttling value set by user. e.g. a throttling value of 0.7 means that during each 100ms, the worker thread must be sleeping for 30ms and process the job for 70ms.

```
time.sleep((100 - self.throttling) / 1000.0)
```

c) Hardware Monitor:

The hardware monitor is responsible for collecting information about the hardware. It monitors CPU utilization information (`psutil.cpu_percent()`) and user throttling value.

```
""" Performs a system call to record another sample of cpu utilization. """
def record_cpu(self, scheduler=None)

""" Calculates and returns the average CPU usage over a specific period. """
def get_cpu_usage(self)
```

We implement an interface that allows the user to dynamically specify a throttling value during execution.

```
""" Interface for throttling the worker thread through the hardware monitor. """
def throttle(self, value)
```

d) Transfer Manager:
The transfer manager is responsible of performing a load transfer upon request of the adaptor. We implement two thread function, one is to move jobs from the job queue and send them to another node: `send_job(self, job)`; one is to receive any jobs from another node and place them in the job queue: `receive_job(self)`.
Here we use UDP to transfer jobs.

```
self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.socket.bind((HOST, PORT))

""" Send job to another node """
job = self.job_queue.get()
self.socket.sendto(data, (self.remote_ip, PORT))

""" Receive job from another node """
data, _ = self.socket.recvfrom(8192)
self.job_queue.put(job)
```

e) State Manager:
The state manager is responsible of sending and receiving state from another node. Similar to transfer manager, we implement two thread function, one is to send local state information to remote node: `send_state(self)`, one is to receive remote state information: `receive_state(self)`. We update the state information every 30s.
Here we use UDP to transfer state information.

```
self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
self.socket.bind((HOST, PORT))

""" Send local state to another node """
self.socket.sendto(state_p, (self.remote_ip, PORT))

""" Receive remote state from another node """
state_p, _ = self.socket.recvfrom(2048)
```

f) Adaptor:
The adaptor is responsible for applying the transfer and selection policies. Transfer policy is set as a comparison between the local and remote nodes. When the size of the remaining job queue for each node differs in size by more than a threshold value, say 30 jobs, that differential will be transferred to the node with fewer jobs. We use a sender initiated transfer, which has less overhead but is less responsive than a receiver or symmetrically initiated transfer.

Step 3: Aggregation Phase
After all jobs are successfully processed, our system aggregate the result into remote node and display the result. We accomplish this by sending finished jobs from the remote node to the local node and requeueing completed jobs in the master node. In this way, all completed jobs will be stored in the original master node job queue.

Step 4: Further Improvements

a) We implement three transfer policies: sender-initiated transfers, receiver-initiated transfers and symmetric initiated transfer. The analyze detail is listed at Part3

```
def sender_init(self):
""" Initiate the transfer job based on local state """

def receiver_init(self):
""" Initiate the transfer job based on remote state """

def symmetric_init(self):
""" Apply both sender and receiver init """
```

b) We compress the data when sending to save time and bandwidth. Here we use zlib module in python.

```
# compress data
data = zlib.compress(job_p)
# decompress data
job_p = zlib.decompress(data)
```

c) For Bandwidth and Delay, we do not think think it will influence the performance of our load balancer. Due to compression, each job only has 900 bytes, and the socket sends out a job every 0.15 second. Therefore, out load balancer only consumes 80 kbit/s, which is way lower than the bandwidth limit 10mbit/s. The only time that this delay will come into effect is at the end of program execution, when a node will run out of jobs to execute while the jobs are being transferred to the other node. We avoid this problem by only sending half the remaining job queue length of the node during our last load rebalancing transfer.

d) We implement a GUI:

- Start the launcher bootstrap phase
- Change the throttling
- See the local state info
- See the message list

e) We implement multithreading in the systems. Because python's Queue is a thread safe structure and each job completes independently, this is simply a matter of allocating multiple worker threads. We modified every function call that throttles, checks state and creates the worker thread to instead operate on this list of threads. For maximum performance gain on the local machine and minimal overhead on the remote machine, we utilize 4 worker threads.

Part 2: Details of how to run your program

Step 1: Update the config.json with the ip addresses of your local and remote nodes. Example:

{"master": "172.22.156.12", "slave": "172.22.156.72"}

Step 2: <On Remote Node>

python launcher.py S

Step 3: <On Local Node>

python launcher.py M

Step 4: Wait for jobs to finish and observe result.

Part 3: Analysis

We tested the program with 1024 * 1024 vector, 1024 jobs, and 100 throttle single worker thread. The JOB_QUEUE_MAX is 100, JOB_QUEUE_MIN is 50. State will be updated for every 2 seconds.

The following is the raw data we got

Sender

It take 100.962810 seconds to finish all jobs

512 job sent

512 job receive

Pending job [1020, 743, 497, 482, 468, 454, 439, 425, 410, 396, 381, 367, 353, 338, 323, 308, 294, 279, 264, 249, 235, 220, 205, 190, 176, 161, 146, 132, 117, 103, 88, 86, 85, 85, 86, 85, 86, 87, 85, 72, 59, 46, 33, 20, 7, 0, 0, 0, 0, 0, 0]

Receiver

It take 97.745706 seconds to finish all jobs

512 job sent

512 job receive

Pending jobs [1020, 648, 492, 477, 463, 448, 433, 420, 406, 391, 377, 362, 347, 333, 318, 303, 288, 274, 259, 244, 230, 215, 200, 185, 171, 156, 142, 127, 112, 98, 84, 69, 55, 41, 35, 36, 36, 37, 37, 36, 37, 37, 37, 26, 13, 0, 0, 0, 0]

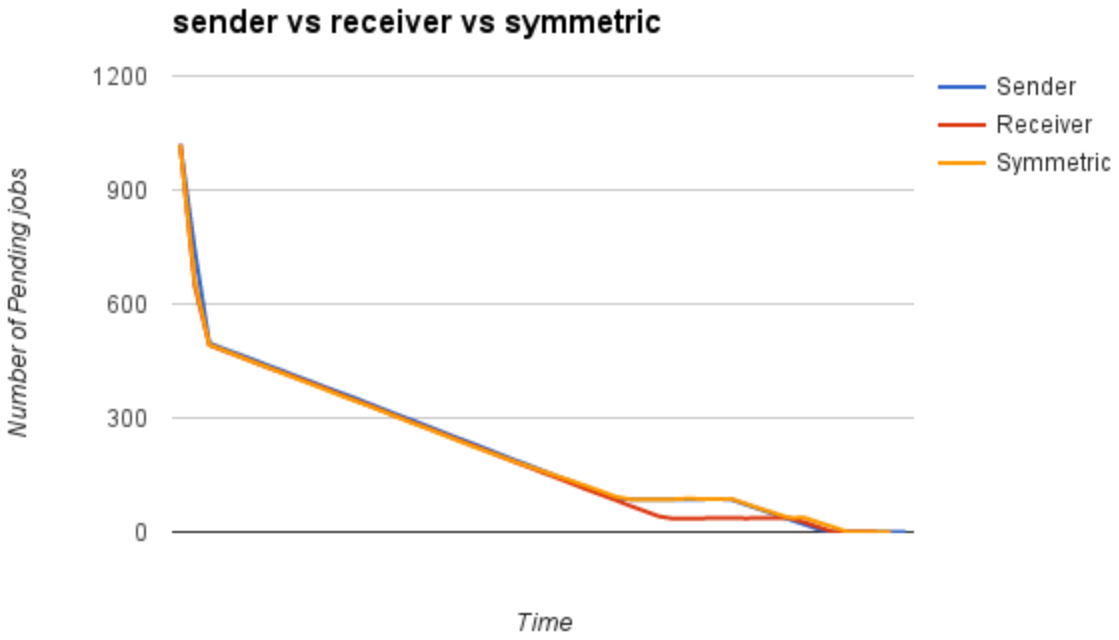
Symmetric

It take 98.816289 seconds to finish all jobs

512 job sent

512 job receive

Pending jobs [1019, 661, 493, 478, 463, 448, 434, 419, 404, 390, 375, 360, 346, 332, 317, 302, 287, 273, 258, 243, 228, 214, 200, 186, 173, 160, 147, 134, 121, 107, 93, 86, 86, 86, 86, 88, 87, 87, 87, 74, 61, 49, 37, 38, 25, 13, 0, 0, 0, 0]



Discussion:

In the sender initiated transfers the over-loaded node (sender) initiates the transfer while in the receiver initiated transfer the underloaded node (receiver) initiates the job transfer process. Symmetrically initiated transfers have both the components. The idea is that during the during low system loads the sender initiated component is more successful while during high system loads the receiver initiated component is more successful because the polling done from the receiver node is more successful in this case. In our case the different transfers showed a similar performance may be because of the fact that we had only two nodes and the transfer policy was looking at the number of jobs present in the queue and initiating the transfer based on that.