The Design of a Centralized Load Balancing System for a Homogenous Cluster

An Overall System Design Demonstrating Efficiency and Scalability

Nicholas Adamou
Computer Science
Cornell College
Mt. Vernon IA United States
nadamou20@cornellcollege.edu

Bingzhen Li
Computer Science
Cornell College
Mt. Vernon IA United States
bli21@cornellcollege.edu

Jillian Shew
Computer Science
Cornell College
Mt. Vernon IA United States
jshew20@cornellcollege.edu

ABSTRACT

The goal of this system is to design and implement a distributed centralized load balancing system for a homogeneous cluster, and therefore the architecture, as well as the design and the security of the system, must be given careful thought and consideration. The system at its core must be able to do the following tasks: 1) Add new node to the cluster, 2) Maintain a list of nodes currently active within the cluster, 3) Delegate a job to the most optimal node in the cluster, 4) Determine the most optimal node based on its CPU Utilization at a given time, 5) Request CPU Utilization from all active nodes to aid in the determination of the most optimal node, 6) Delegate tasks sent from a given client node to the optimal internal slave node in the system and communicate back to the client whom sent the request. Because of these tasks the system is required to perform, the goal of the overall system design must be taken from the perspective of efficiency and scalability. A load balancer must quickly and efficiently handle application traffic across a number of servers. This allows for increased reliability of client connections within the system and thus, allows a system to scale as its user-base increases.

1 Design Overview

Given the nature of such a system that is our goal to create the most optimal architecture is master-slave because each node in the cluster must both act as a client and server concurrently in order to achieve the desired results of the said system as seen in Figure 1. However, we have chosen to maintain a centralized master-slave architecture rather than a decentralized system because of its efficiency, consistency, maintainability, and scalability. Having one node act as a reverse proxy for its clients rather than each node providing this said task has its benefits in that its simple and easier to implement and maintain over its counterpart. Additionally, because of this aspect, all nodes must first connect to this said centralized node which makes it easier to track and maintain the connections across the cluster. This improves scalability with the respect that we can simply just add another node to the cluster and the system should inherently handle the additional node. Centralized systems do have their downsides however; for example, it suffers from a single point of failure. If the central—or master—node in the cluster goes down, the individual "slave" machines attached to it are unable to process requests and send their output back to their source. However, on the flipside, decentralized networks require more machines, which means more maintenance and potential issues. Moreover, the implementation of the said network is much more difficult than a centralized system. In short, given the short time-span provided to us to complete this said project and the aforementioned advantages and disadvantages of centralization, the implementation of a centralized master-slave system was chosen.

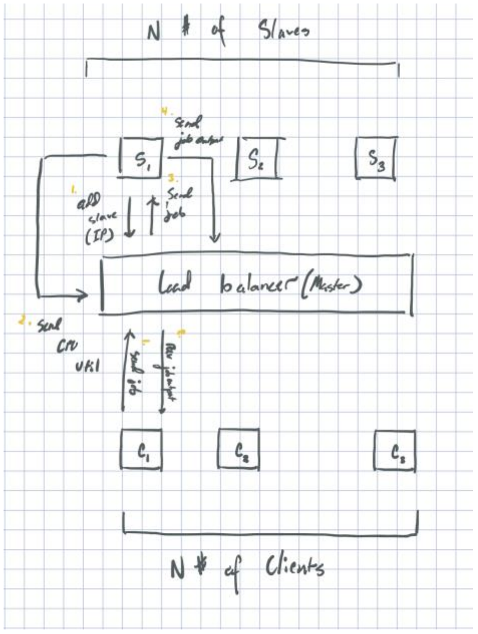

Figure 1 System Design Overview

1.1 Master

The master node in the cluster acts as the centralized node whereby all nodes communicate with. Essentially, it acts as a reverse proxy between the client nodes and the slave nodes. Because of this, every client node is not aware of any of the slave nodes and every slave node, is not aware of any client node. Thus, the master acts as an intermediary between the clients and the slaves nodes. The master has 7 different, yet distinct jobs: 1) Add a new node to the cluster, 2) Listen for CPU Utilization values sent from nodes within the cluster, 3) Maintain a determination of the most optimal node in the system based on each node's CPU Utilization at a given time, 4) Listen for incoming client connections, 5) Process client connections, 6) Send a job to the most optimal node in the cluster and wait for the output of said job from the node the job was delegated to, 8) Send job output back to its associated client.

1.1.1 Selecting the "*Most Optimal*" Slave

Selecting the most optimal node is a crucial task for maintaining the performance of the system. Thus, it is important that we define what most optimal means with respect to this system. Most optimal in the strictness sense is a slave that maintains the least CPU utilization across all other slaves in the network. Thus, a sorted array or list of slave nodes is defined as the list of nodes whereby each node is in increasing order based on its CPU Utilization. The priority of a given slave is based on its CPU utilization at that given time; the higher the number the lower the priority and visa versa. An unsorted list of slave nodes can be defined as strictly the opposite whereby the order is determined through FIFO. However, the data structure, as well as the technique chosen to sort such a network is a task that when not done properly can lead to inefficiencies and unwanted overhead within the system. There are three different techniques that accomplish this task, which is discussed in the following three subsections. However, with respect to our system, we have elected to implement the technique discussed in Section 1.1.1.1 for the reasons outlined in said section.

1.1.1.1 Unsorted array + update *Optimal Slave*

This technique utilizes a generic array of unsorted slave nodes. During each pass of the listen for CPU Utilization thread, we maintain the optimal slave by comparing its CPU Utilization to that of the slave whom is sending its CPU Utilization value. If its CPU Utilization is lower than that of the optimal slave, then we update the optimal slave to be that of the sender slave.

One of the major advantages of implementing this method is that it is extremely flexible. We can quickly and easily adjust the optimal node based on new CPU Utilization data provided by the slave nodes as they send their CPU Utilization to the master node. This does not require the use of traversing all the nodes and determining which is the minimum, as it is dynamic. Thus, this allows for one less thread to be implemented on the master side as discussed in Section 1.1.1.3 and does not require the internal logic to maintaining a min-heap.

A disadvantage of this method is that when compared to that of the min-heap implementation as discussed in Section 1.1.1.2, it requires additional steps during each pass to ensure that we have the node that has the minimum CPU Utilization. A min-heap, ideally, would allow the system to both add the slave node into the system and put the node in the correct order based on its CPU Utilization in one step. However, based on the logic of the system, when a slave acknowledges its presence to the master node, it does not send its CPU Utilization, but rather sends its IP address. Thus, the min-heap would have to be maintained during the same step that we maintain the minimum node as discussed in Section 1.1.1.1.

1.1.1.2 Min-heap (insert into sorted order)

This technique uses a min-heap to maintain the optimal slave. A min-heap is a binary tree such that the data contained in each node is less than (or equal to) the data in that node's children. This allows the root node to contain the most optimal node at that given time.

A min-heap is algorithmically efficient such that the minimum element is contained at the root of the tree and thus would require O(1) time to retrieve. In addition, the memory usage of such a structure is minimal when compared to the other two methods. Thus, it requires no additional memory space to sort correctly. Moreover, a min-heap exhibits consistent performance meaning that it performs equally well in the best, average, and worst cases.

As discussed in Section 1.1.1.1, a min-heap is optimal for the reason that it ideally requires one less step than the other methods discussed in Sections 1.1.1.1 and 1.1.1.3. However, given the logic of the system, as discussed in Section 1.1.1.1, this is actually untrue because of the aforementioned reasons. It requires more overhead than the other two techniques. Moreover, it requires additional sorting time as it is an unstable sort. A stable sort maintains the relative order of items that have the same key (i.e. the way they are present in the initial array). Heapsort is an unstable sort because it might rearrange the relative order of the nodes it contains in order to maintain its internal heap property.

1.1.1.3 Two arrays (insert + sort concurrently)

This technique maintains two separate yet distinct arrays of slaves where one contains an unsorted list of slave nodes and the other contains a sorted list of slave nodes. The sorting operation occurs concurrently with that of the insertion.

One advantage of this strategy is that the sorting logic would not need to occur within the logic of the request for CPU Utilization from all slaves as does the method discussed in Section 1.1.1.1. It would occur asynchronously to that of said operation. Ultimately, allowing for increased readability because of the existence of a separate thread that's sole purpose is to sort the list of nodes.

The major disadvantage of this method is that it requires the implementation of two threads on the master-side. In addition, the space required is drastically larger than that of the previous two methods discussed in Sections 1.1.1.1 and 1.1.1.2 because of the need for two arrays of slave nodes. Moreover, there would be more need for control between either asynchronous threads that the other two methods do not require. Also, it is not efficient as this would incur a massive overhead on the part of the master and thus puts more load on the master than that is needed.

1.2 Slave

The slave nodes are the individual hosts or other computers in the cluster that act as the workers of the system. They receive jobs, execute them, and report their output back to its source. They have 4 different, distinct jobs: 1) Connect to the master node to acknowledge that it's alive and able to receive jobs 2) Send CPU Utilization to master every N amount of time 3) Listen for a job sent from the master node and add it to its job queue 4) execute each job that it was delegated in FIFO order and respond back to the master with the output of the completed jobs.

It's important to consider how often the system receives each node's CPU Utilization as it can have an impact on the overall system load with respect to the number of nodes in the cluster. If hundreds of slave nodes send their CPU Utilization every second, there would be a greater failure rate and load on the part of the master node. For this reason, it is imperative that we consider this factor with great care and consideration. Thus, in our system, we set a maximum random sleep time of 10 seconds to allow for a greater change in CPU Utilization. Moreover, the sleep time is random because it allows for variance in returned CPU Utilization values.

1.2.1 Master Slave Communication

Communication between slave and master begins with the individual slave nodes interacting with the master node as depicted in Figure 2.

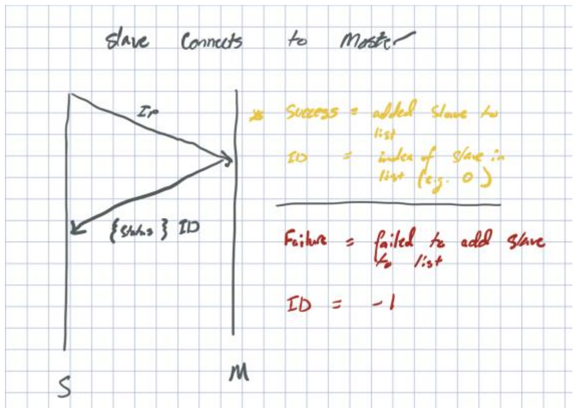
Slave Connects to Master
IP
{Status} ID
Success = added Slave to list
ID = index of slave in list (e.g. 0)
Failure = failed to add slave to list
ID = -1
S
M

Figure 2 Slave connects to master

This interaction notifies the master of the slave and adds the slave into the cluster. The slave starts this interaction by connecting to the open socket on the master for accepting new nodes and submits its IP address. The master will then respond back with the status of the operation to add the slave to the cluster. If the operation succeeded, the master node will return the slaves identifier inside its internally managed list of nodes along with the status of the operation. If the operation failed, however, the status message will indicate this failure and the id of the slave will be -1.

Figure 3 showcases the interaction that occurs when the slave sends its CPU Utilization value to the master node. As discussed in section 1.2, this interaction happens at most every 10 seconds. This interaction is crucial to the system because the system is unable to operate without the knowledge of a slave's CPU Utilization value. This interaction begins with the slave connecting to the master's open port for accepting CPU Utilization values. The slave will then send its id as well as its CPU Utilization value to the master. Depending on whether or not the operation to update the slaves' CPU Utilization value is successful or not, the master will respond with the appropriate status to the slave.

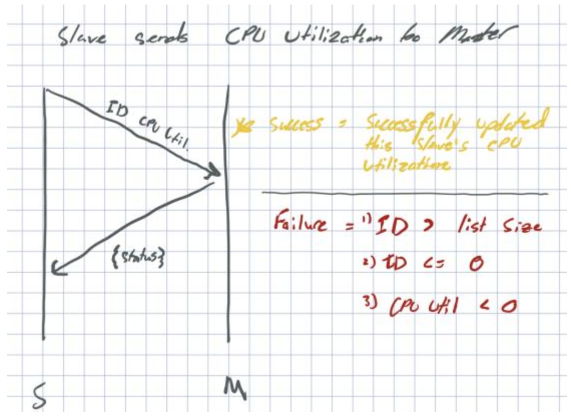
Slave sends CPU Utilization to Master
ID CPU Util
{status}
Success = Successfully updated this slave's CPU Utilization
Failure = 1) ID > list size
2) ID <= 0
3) CPU Util < 0
S
M

Figure 3 Slave sends CPU Utilization to master

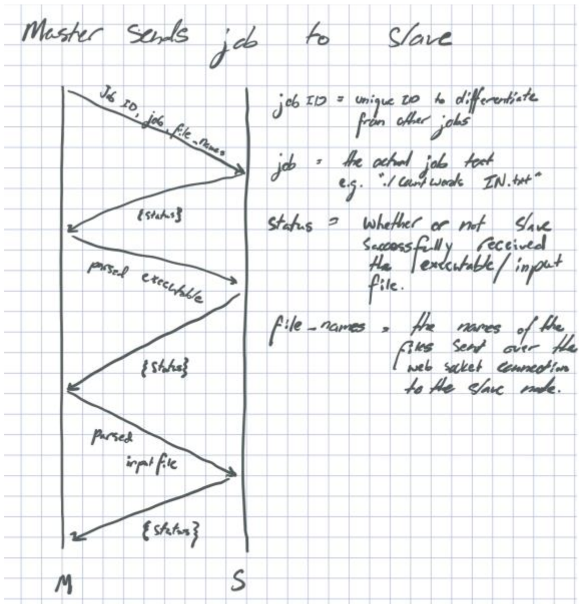
Master Sends job to Slave
Job ID, job, file_names
{status}
parsed executable
{status}
Parsed input file
{status}
job ID = unique ID to differentiate from other jobs
job = the actual job text e.g. "./countwords IN.txt"
Status = whether or not slave successfully received the executable/input file.
file_names = the names of the files sent over the web socket connection to the slave node.
M
S

Figure 4 Master sends job to slave

Figure 4 depicts the interaction that occurs when the master node delegates a job to the optimal slave. This interaction begins with the master sending the optimal slave a message containing the job to complete as well as the corresponding job id associated with this job. The master would also send the names of the binary file as well as the input file to the slave. The file-names are required so that the destination node knows what to call the buffer containing the sent data on its file-system. The slave will then acknowledge the either the successful or failure to receive the sent data. Next, the master would send the parsed binary executable as well as the parsed input file in the form of a buffer to the slave. The slave would then be able to write the contents of the buffer to a file and be able to execute it without any additional permissions. Finally, the slave would respond back to the master with the status of whether or not the task was completed successfully.

Figure 5 showcases the final interaction that occurs in the system is when the slave completes the assigned job and sends it back to its source. This would involve the slave sending back the status of the operation (whether or not it failed or succeed), the id of the job in order to differentiate this job from another, as well as the name of the file associated with the output file. The file-name is required so that the source knows what to call the buffer containing the output data on its file-system. The interaction would also utilize a web socket connection to send the output file of the job to the master node. Once the slave completes the sending of the output of the job to the master node, the master would respond back acknowledging the success or failure to receive the output of the job.

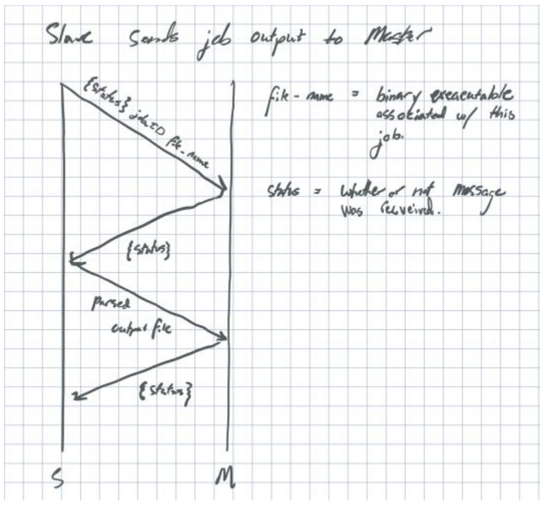
Slave sends job output to Master
{Status} jobID file-name
file-name = binary executable associated w/ this job.
Status = whether or not message was received.
{status}
Parsed output file
{status}
S
M

Figure 5 Slave sends job output to master

1.3 Client

The client nodes are the respective devices that send jobs to the master node and wait for a response containing the output of the job sent. The clients are responsible for launching a job and waiting for its output to return from the master. The clients are not aware of the master's slave or worker nodes. It can only communicate with the master node. As discussed in section 1.1, the master node will handle the delegation of the job to the best worker or slave node and getting the output of said job back to the source client.

1.3.1 Client Master Communication

The client communicates jobs with the master node as seen in Figure 6. The client initiates the communication with the job request data. This data contains the name and size of the files that will be transmitted for the slave node to complete its task. In order to minimize the traffic across the network with regards to sending the binary executable and input file for the job, we elect to send each file in chunks. This would significantly reduce the packet wastage and improve the speed of the transfer and thus the performance of the system. Once all of the chunks have been received on the destination side, it will acknowledge the successful retrieval of the data back to its source client. Then, the system will execute the task on the most optimal node and will send back its output in the form of a text file back to the client.

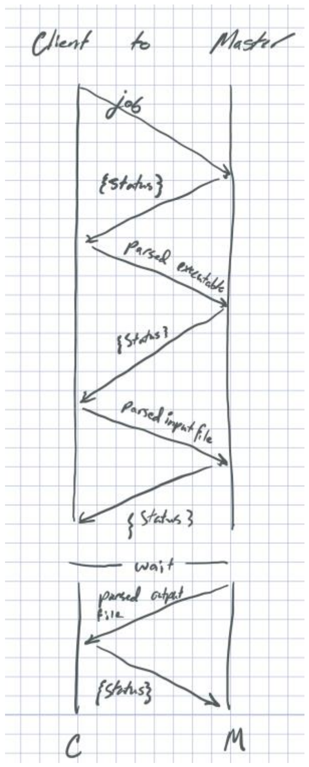

Figure 6 Client sends a job to master

3 Conclusion

Given the goal of the system and our design decisions, we believe that we achieved the objectives for the system. Even though each algorithm that was discussed in the previous Sections had their inherent advantages and disadvantages for choosing the most optimal node within the system, we believe the techniques discussed in Section 1.1.1.1 as it relates to its advantages outweigh its disadvantages. A min-heap might be algorithmically advantageous, as discussed in Section 1.1.1.2, but given the design constraints, maintaining an unsorted list of nodes and updating the optimal node during each pass of the request for CPU Utilization thread would be more ideal because it would ultimately incur less overhead than the other two methods discussed in Sections 1.1.1.2 and 1.1.1.3.