Project 2 Report

Ayush Vora
arv1924@rit.edu

Harshal Khandare
hvk1162@rit.edu

Pranav Dadlani
pad2039@rit.edu

ABSTRACT

The evolution of distributed computing over the recent years has given rise to an array of novel approaches for solving any task of huge magnitude. In this paper, we propose one such solution which focuses on designing a distributed computation framework capable of performing sorting and averaging tasks on a large dataset of numbers. Designing an architecture for such a system requires careful examination of the available resources and developing an exquisite communication protocol offering scalability, performance and robustness. Our design was structured to fulfill these requirements. It aims at providing a seamless communication environment which is capable of handling increasing workloads and exhibiting exceptional fault tolerant capabilities. In this paper, we discuss our entire design process,a complete overview of the implementation and a detailed analysis of the experiments performed.

1. INTRODUCTION

The nature of distributed system environment available and the task assigned are two important factors to be considered while designing a distributed computing framework. In order to perform a large operation more efficiently and with a greater performance, it is essential to distribute or share the workload among multiple systems. We had a set of 6 Raspberry Pi's which were connected by a reasonably fast network. One Pi acts as the master node which handles discovery, registration and monitoring of worker nodes. It also performs preparation and distribution of jobs to the worker nodes. Other 5 Pi' are the slave (worker) nodes which perform computations of smaller jobs. Overall, all the Pi's work cohesively towards solving the problem. The use of such a distributive computing environment simplifies the huge task by utilizing the computation power of a set of slave Pi's. In the next section we discuss our initial design idea followed by the enhanced designed which we have implemented. We then talk about some of the tweaks performed to our design to improve its performance. Further, we explain our system implementation and provide the experiment details along with its analysis. We finally conclude by listing the limitations of our design and discuss possible future improvements.

2. DESIGN

2.1 Initial Design

The initial design is based on a master-slave architecture. The master would be responsible for distributing jobs to slaves and slaves would perform their jobs and return results to the master. Finally, the merge step would be performed at the master which would combine all the results to give a final result.

We started with the most basic design where the huge file is divided into number of chunks equal to the number of slave Pi's. The aim of such a naive design was to serve the basic functionality needed. We had little to no idea about the size of the file or the network transmission speed and hence thought that this design would be feasible. After getting some input on the resources available at our disposal and figuring out the performance constraints for each of the components such as the processing and transmission speeds, we realised that this basic design had several limitations.

The first and the most important limitation was with the size of each individual chunk. The size of each chunk would become enormously large with the increasing dataset file size as we are limited by the number of slave Pi's. Also, transmitting such huge chunks across a network with limited bandwidth would be a time consuming process. This means that the slave Pi's will have to wait for the entire chunk to be received before they can start processing and return the solution chunk back. This would also mean that the Pi's processing capabilities wouldn't be fully utilized. Under fault tolerance conditions, the performance would degrade even further as it would require the master Pi to retransmit the entire chunk. Even though this design might perform better under perfect conditions, it would fail miserably in a realistic setting.

Our thought process was headed towards overcoming the above mentioned design issues. As the transmission bandwidth was a bottleneck for large sized chunks, we decided that its size should not cross a certain predefined limit. The limit should be set such that both transmission and processing of a data chunk could perform simultaneously in an efficient manner. We decided that the focus of our design should be in maximizing the performance of each available resource and hence the entire system by exploiting their capabilities to the fullest. A 'fast-come-fast-serve' system

where all resources perform by complementing each other to arrive at the desired state was something we were looking for.

2.2 Proposed Design

The proposed design was developed keeping all the limitations of the initial design in mind. One of the requirements we realized is that the implementation should not be dependent on the size of the dataset rather the size of the chunk for any dataset should be modifiable depending on the network bandwidth and processing capabilities. In the proposed design, we decided to keep a fixed sized chunk and the size of the chunk would be determined on the basis of empirical analysis. Each chunk corresponds to a job and all jobs exist in a queue at the master.

The most important modification to the initial design is, rather than master forcing the slaves to process jobs, the slaves would voluntarily request for jobs from the master as per their own availability. This modification simplifies the coordination of jobs between various slaves in distributed environment which are prone to frequent node failures. It also considers whether a slave is actually available for efficient processing for jobs. With this design a master Pi is not required to decide which slave needs to be given a failed job. It would simply queue a failed job and when a slave requests for a job it can be processed as per its position in the queue. This design thus guarantees that all jobs are processed sooner or later without having a complex job distribution protocol in case of failures.

With this design, size of the chunk is determined by keeping in mind the fault-tolerance. As the size of each chunk becomes smaller, the fault tolerance performance improves. The probability of a chunk being dropped during transmission is inversely proportional to the size of the chunk(more the probability lesser the size of the chunk). This probability is determined after some empirical analysis.

3. DESIGN PROCESS

3.1 Generating Jobs

We started by designing a Job generator first. A naive Job generation approach would involve waiting a long time before jobs are ready for processing.The processing power of the master Pi is a severe bottleneck in performing IO related tasks. Hence, a better approach was required to handle this constraint. To make the processing and transmission concurrent, we introduced the concept of a Job Queue where new jobs are enqueued and already present jobs are dequeued and sent to slave Pi's for processing. This queue behaves similar to the producer-consumer problem where both the enqueuing and dequeuing of jobs can take place concurrently. This way the master Pi does not have to wait for the all the jobs to be generated before it can start transmitting them. It can start distributing jobs as soon as they are generated. Using such a job queue enhances performance and having a producer-consumer paradigm supports a distributed environment where concurrent processing is favorable.

3.2 Distribution of Jobs

Jobs are distributed to the slave Pi's in response to the job request sent by the respective slave Pi. A slave Pi requests for a job as soon as it is free and ready for processing. The master handles each of these requests made by multiple slaves by dequeuing an available job from the job queue and transmitting it to the requesting slave. If there are no available jobs, the master ignores this request.

3.3 Handling of Jobs

The generated jobs being a list of unique integers are given to slaves following slave requests. The Master eliminates duplicates from the dataset. For the given task, the slave would first sort these unique numbers using Radix sort. Radix sort is a non-comparison sort. It was chosen because for numbers which lie within a smaller range and have fewer number of digits, it would have a better performance than merge sort. For example, suppose there are $N = 2^{32}$ numbers to sort, which is about 2 billion numbers the time complexity for merge sort would be $O(NlogN)$, which is $O(N(32))$ while the same number of numbers having 10 as the maximum length of digits for a 32 bit integer in base 10 can be sorted in $O(N(10))$ time.Since for 32 bit integers, 10 would indeed be maximum length of digits, it would be better to choose radix sort instead of merge sort for large number of integers to sort. Radix sort thus, would always perform better when there are more number of numbers to sort but have a smaller digit length. Since the list of unique integers must also be used for the second task, we compute a local average of these integers. After sorting and computing local average for this list, the sorted list, local average and size of the list, are sent back to the master or any other pi willing to perform a final merge. For our design, we choose to use the master to perform the final merge, because we wanted to use as much resources we could for processing all the jobs. The merge operations would give the final sorted list and the global average for the given dataset.

3.4 Communication

The communication protocol design involved the use of sockets. Initially we planned to use a single socket connection with multiple data streams for incoming traffic and a similar kind for outgoing connection from the Master Pi. After performing a few trials with such a protocol, we understood the nature of sockets. We learned that at a given time only a single stream may be active and the master had to wait until it could start sending data on a different stream. This would make the transmissions a sequential process, with every slave Pi waiting for its stream to become active. This was againsts our proposed concurrent approach. We wanted all the slave Pi's to be actively involved with the master Pi. The single socket design makes an otherwise available slave Pi inactive due to the inability of using multiple streams over a single socket.

To solve this issue and align the communication protocol with our principle of concurrency, we decided to use multiple sockets each having a single data input and output stream. Here, every slave Pi will have a dedicated socket connection with the master Pi. This way each slave Pi could independently communicate, send and receive data with the master Pi using their own dedicated data streams. As there is no dependency of streams across different sockets, simultaneous transmissions can take place and the available slave Pi's could be served by the master immediately.

3.5 Fault-tolerance

As each job will be associated with a particular socket, in case of failure of a slave Pi, the disconnecting socket could first place the job which was being processed back to the Job queue as it is incomplete. Since the incomplete job is back in the job queue, it will be eventually requested by a slave and will get processed. Thus whenever a socket stream connection will fail the job associated will never get lost and is guaranteed to be processed sooner or later. Hence this design is fault-tolerant design with easy job recovery features.

4. SPECIAL ENHANCEMENTS

We integrated certain tweaks to our design in order to efficiently enhance the performance of our system. Following are the important enhancements

4.1 Slave Job Queue

To achieve concurrency at the slave Pi, we introduced the idea of Job queue of free slave threads. The size of this queue is 3 which means that the there could be 3 concurrent processes running simultaneously at the slave. It could be thought of as 3 slaves within a slave Pi. We strategically start a new slave thread for processing such that at any given time one thread is processing the data (sorting), another thread is receiving new data to be processed from the Master and the third thread might be returning the solution of some previous data. Hence, we decided that keeping 3 slave thread could handle the processing, sending and receiving of 3 different chunks. This way the slave Pi is never free and constantly working.

4.2 Unique ELements

Instead of sorting the original dataset as it is, we decided to sort only the unique numbers. As the numbers dataset is randomly, it is bound to contain several repeating numbers. We initially store unique number along with their frequency in a map and prepare job chunks having only unique elements. Doing this greatly reduces the size of data to be transmitted across the network. Only unique elements are sent as chunks for sorting and their frequencies are used while writing the final solution at the master.

4.3 Resusing Socket Connections

We are making dedicated socket connections for each slave threads with the Master. This means 3 socket connections for each slave Pi. So, instead of creating a new connection before starting a communication, we create, connect and the store the sockets for each slave Pi at the master. We reuse these connections for later communications. Thus, we are avoiding the overhead a creating and closing a connection for each communication. We are only creating a new connection while recovering from a socket disconnection due to failure at slave Pi.

5. IMPLEMENTATION

We have modularized our entire design into various components. Each component is concerned with a specific task and works in unity with all the other components. We describe all the components in this section

5.1 MASTER COMPONENTS

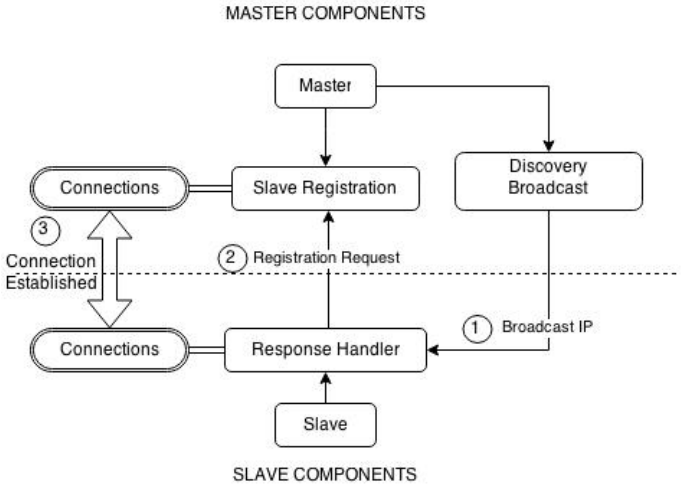
5.1.1 Job Generator

The job generator is a thread that runs as soon as the master Pi starts. It creates jobs for the slaves. These jobs are a list of unique integers. The actual list of integers is first read from the datasets and placed in a hash map, with key as a unique integer and value as the frequency with which the integer has occurred in the file. As and when a unique integer is encountered it is place in the hash map as well as an Array List. If it is a duplicate integer its value is incremented in the hash map. The array list is then serialized and placed in the filesystem as a .uvc file which stands for unique value chunk. The idea behind using the hash map is to reduce the size of the chunk that is sent to a slave pi. Each created file uses the start index of each unique integer list chunk as its filename. Thus when this job needs to be sent again to a slave in case of a failure, it can easily be identified and transmitted. The chunks are serialized and persisted in the file system so as to decrease the amount of main memory usage. As each chunk list file is created it increases the 'created jobs counter', also their filenames which are indexes are place in the job queue, which is just a queue of integers, one at a time.

5.1.2 Discovery Broadcast

This is the Discovery Broadcast thread. Master will periodically broadcast its own IP to all the nodes in the network. It uses datagram packet to send this broadcast. Using a UDP scheme meant that the discovery would be fast. It need not be very reliable since its purpose is to identify available slaves as quickly as possible and not perform actual transfers.

A. Registration Phase



5.1.3 Slave Registration Handler

The slave registration handler is responsible for accepting registration requests from slaves and registering the three server sockets of a particular slave. The slave sends a registration after receiving a discovery broadcast. After receiving a registration request from the slave, an acknowledgment is

sent to the slave which starts listening for a connection on its server sockets immediately. The master initiates a connection with those server sockets. In this way a reliable connection with a slave is established with three dedicated streams which are used repeatedly for successive job request-response between Master and Slave. It stores the IP and port of the slave socket with the corresponding socket in a Hash map called created sockets.

5.1.4 Master Request Handler

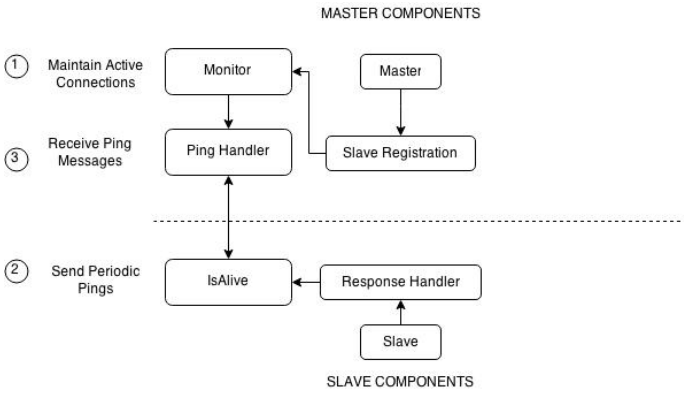
This thread runs immediately as the master is started. Once registration is complete master listens for requests on a datagram socket. If it receives a request, it retrieves the socket related to the slave IP and port embedded within the request from the hash map created sockets, it checks whether jobs are available in the job queue and the socket is not currently in use. If conditions are satisfied then it sends out an acknowledgment to the slave and starts a new thread called chunk send receive which is associated with a particular socket connection, a particular job and the corresponding solution. Once it has started this thread it immediately dequeues the job index from the job queue. It also places the IP and port for the request in a hash set that indicate the currently used stream connection. The reason for keeping this thread is to have a controlled approach to handle all requests. It will keep on listening for request until the entire file is read and the created jobs counter equals the completed jobs counter.

5.1.5 Chunk Send Receive

This is a thread thats gets started when a job request can be serviced by the master. It independently handles the transmission of a particular job chunk through an associated unused socket stream with a particular pi's socket connection and also handles the reception of the result that is to be sent by the slave pi through that same socket connection. This thread handles deserializing the arraylist object in the .uvc file for the associated job index in the job queue and sending it to the slave. Once the slave has finished processing, it sends it to the master and this thread handles receiving it from the slave and reserializing the processed object. Once it has done so it removes the associated IP and port for this socket connection from the hash set. It also increases the completed jobs counter. Now if any the socket connections fail this thread catches the exception adds the corresponding job index back to the queue and removes the socket from the created sockets hash map. In this way all threads associated with a chunk can recover the chunk.

5.1.6 Slave State Monitoring

It monitors the currently alive sockets by keeping a currently alive sockets hash map. First it checks which sockets are currently alive if they have false values in the hash map they are added to the list of currently dead sockets and eventually removed from the currently alive sockets hash map. Every 10 secs it makes all the currently alive sockets values in the hash map as false. All the slave sockets must then ping the ping handler to make the values for their currently alive sockets as true to indicate that they are alive. C. Monitor



5.1.7 Ping Handler

It receives pings from the currently alive slaves within a time frame of 10 secs and updates the currently alive sockets hash map by making the corresponding values true.

5.1.8 Merge

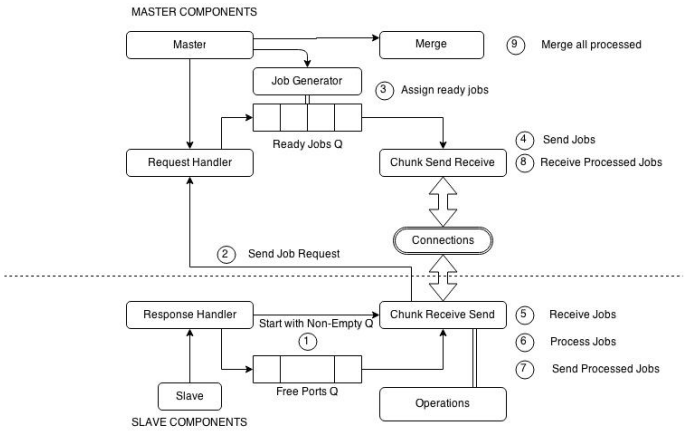
It merges the various solution files received from the slaves to get the final result. For the first task which is sorting we use a Sort m-lists algorithm which sorts m sorted lists to get a final sorted list in O(mN) time. For the second task the list of local averages are combined to get a final global average.

5.2 SLAVE COMPONENTS

5.2.1 Slave Response Handler

This thread receives the discovery broadcast from the Master and sends a registration request. It also starts an IsAlive thread. If it does not receive an acknowledgment for registration within a certain time then it resends the registration request. Since the master is only one node, losing registration request is a certain possibility hence retransmission of request is implemented for that purpose. Once it receives an acknowledgement it starts listening on its server socket ports for connections from the master. Once all connections are established, it starts a chunk receive send thread. This thread will monitor the state of each slave socket, if a slave socket fails, it assumes that there is a fatal connection error and starts a new response handler slave thread.

B. Request Response Phase



5.2.2 *Chunk Receive Send*

The chunk receive send thread is a thread which is responsible for sending a request to the master after ensuring that the queue of available socket port connections is not empty, on a particular slave. Since each job is associated with a socket connection, one job can consume one socket connection at a time. If a socket connection is available it will send a request for a job and start receiving on this socket. As soon as it receives a job chunk from the master, it starts a new thread of itself i.e. a new chunk receive thread, while at the same starts processing its currently received job chunk. Hence it can now receive and process in a multi-threaded concurrent manner. Thus this thread provides an enhanced concurrency feature. After receiving the chunk, the chunk is processed by a radix sort algorithm and also the average is computed for this chunk. Both these elements are then sent to the master. This thread thus handles the reception, processing and transmission of the solution file. If there is some IO error or socket error, or a timeout has occurred. It makes the state of slave to indicate as 'bad' and the response handler can appropriately handle it.

5.2.3 *IsAlive*

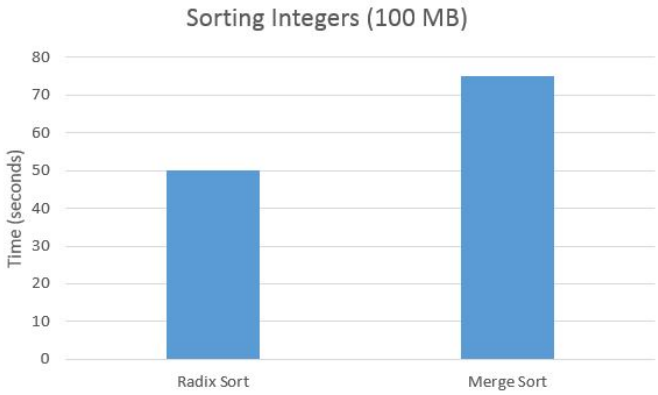
It periodically pings the master's datagram socket connection using the ping handler thread.

5.2.4 *Operations*

Performs various operations such as sort, or compute averages of integer lists.

6. EXPERIMENTS AND ANALYSIS

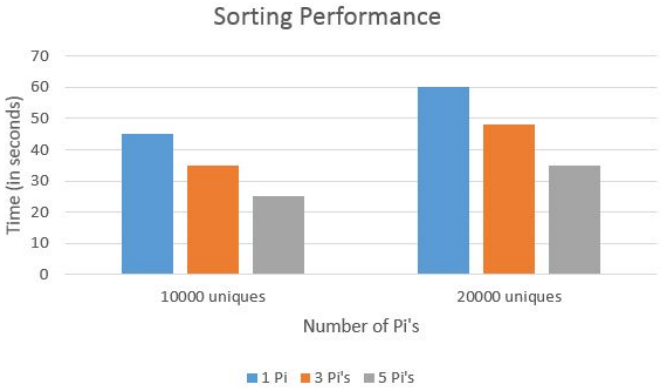
We performed initial tests to analyze the transmission and job processing performance on the Pi's. We used a single slave for processing a huge chunk of 100 mb just for experiment and found that its transmission took 40 sec and sorting took 50 sec using Radix Sort and over 1 min for Merge Sort. Hence, we decided to use Radix Sort within our design. Increasing the number of Pi's, reduces the overall sorting time, as the task is now distributed, but the transmission time is still remains same. We concluded that the transmission of data is a bottleneck which would cause a lot of waiting time before any processing can be done.



We then experimented with very small chunks of data between 500 to 5000 lines in each chunk. We found that such small chunks provide much concurrent capabilities as transmitting a small chunk will be quick and the slave Pi threads could immediately start processing while in the meantime another job is on its way to the slave Pi. Setting the value of chunk size to 1000 lines, we observed that each slave Pi was handling at least 2 jobs at any given time. This would not have been possible if the chunk size was large as it would take one chunk a lot of time to reach only one slave Pi. Smaller chunks travel fast and hence can reach many slave Pi's in a short time. Hence, we finalized a value of 1000 lines as the chunk size as it offered better concurrency.

We also tested our final design under different settings. We initially tested our design using a single slave Pi, with a sample dataset of 1M numbers. There were 10000 unique elements which we would be sorting using the slave Pi's. The entire process took around 45 sec to complete. Increasing the number of Pi's, we found that the overall processing time took around 35 sec with 3 Pi's and around 25 sec using 5 Pi's under perfect settings. We observed that the processing time reduced when the number of Pi's were increased but not by much. This is due to the limited network bandwidth.

We further performed the analysis for much larger files having more number of unique elements. We used 5 Pi's to perform a similar analysis for a file with 1M numbers and 20000 unique elements. The process took around 35 sec as opposed to 25 sec for 10000 elements. As the size of data to be transmitted doubled, the processing time increased. Hence, size of the data to be transmitted majorly affected the performance followed by reducing the number of Pi's. This was clearly visible when we did not use the unique elements approach. It took over 5 mins for processing a file with 1M elements when we sent chunks without eliminating duplicates over the network. This confirms that network bandwidth was a major limitation.



We also experimented with dynamically adding and terminating slave Pi's in between processing. Our system was both adaptable to new incoming nodes by dynamically registering and assigning them available jobs and fault tolerant in the case where a slave was abruptly disconnected. We tested by starting slave Pi's in intervals and observed that the master successfully handled the registration requests sent by new Pi's. This was possible as the master has different threads for handling registration and job requests. Hence, it is capable of processing these two types of requests concurrently. Further, when any slave Pi was disconnected, the master would know that the connection is broken (from a Socket Exception) and will re-enqueue the jobs lost back to the job queue. We did not observe any significant performance degradation from a disconnection and reconnection of a single Pi as the fault handling mechanism quickly kicks in and recovers from the failure.

7. LIMITATIONS

If the number of unique elements to be sorted are too large, the performance of the system degrades firstly due to the network bottleneck and secondly due to the computations required for adding, checking and retrieving elements from the hashmap. The hashmap does not handle very large data efficiently. In such a case a basic approach would be to avoid the initial computations for extracting unique elements and just sending the original dataset as chunks. Our system has defined a general communication protocol for a master-slave architecture and fits well for sorting using both the approaches.

8. CONCLUSION AND FUTURE IMPROVEMENTS

The network transmission and processing was very limited for this task. We managed to overcome these limitations using a concurrent approach principle along with the special design enhancements as explained above. We also realized that for every case within our environment the processing time will always remain lower than the transmission of data, and it may be favorable to perform all the processing on the master itself.

We developed a fast,efficient,scalable and fault tolerant system which was capable of performing huge tasks in a distributed environment. The use of heuristics of unique numbers made it possible for performing this time consuming task even faster. But use of such heuristics do not always perform efficiently. They may be more time consuming under different scenarios. Hence, we have provided both the approaches, one which makes use of heuristics and another which does not. Both approaches follow the same internal design and a general communication protocol. In the future, we plan to make an adaptable system which could decide and switch between the two approaches depending on the environment settings and nature of the task. We could also make our design much more generic by adding several useful components for computing different problems using the same base design.