Protocol Design for Distributed Applications

Data Mining / Volunteer Computing

Rob Brady, Tomas Barry, Calvin Nolan

Introduction:

In this assignment we were tasked to implement a protocol design for distributed applications, a system where users would donate idle time of their computers to computationally intensive efforts known as Volunteer Computing. We created our own platform for volunteer computing using a large dataset of first and last names, where we desired to find if a name was within the dataset by distributing small blocks of data to individual workers and having them process this smaller dataset and then report the results back to the server.

We broke down our project into many parts to distribute amongst our team and to progress the program step by step. Throughout our project we found many obstacles and difficulties to overcome in our implementation and needed much redesign until we came upon our final and current protocol.

Design:

After the initial process of understanding the assignment prerequisite and what it wanted us to create it became obvious that we would need to break down the protocol into smaller tasks that could be completed one by one and ultimately come together to be a functioning volunteer computing program. Our breakdown designed with these problems in mind:

1. Allowing the server to support multiple clients simultaneously.
2. Manipulating and allocating the whole data set to individual clients when requested.
3. Storing and displaying statistics of the process after the program has concluded.
4. Creating sufficient packets to allow the Clients and Server to interact adequately.
5. Lastly, creating a flow between the Server and Clients that utilised all of these functions.

We were given this project with no pre-existing materials to build off of, meaning the entire program was ours to design being given very little direction past what was required for it to achieve. We had some difficulty with implementing the server to support multiple clients due to the heavy use of threading needed to achieve this functionality and the non existent foundation given yet we found a solution to this and many other problems to create our resultant program.

High-Level Description:

For this section we will describe in detail each of the five main design points mentioned above to help fully understand our program and the choices we made in creating it.

Multiple client support:

We use a simple system for connecting multiple workers to the server. Whenever a worker tries to connect to the server, it tries connecting to the default port on the server. If the port address is in use, it increments the address number and tries again. When the worker does eventually connect, the server adds it to the list of connections. When it gets added to the *connectionList*, it adds the address of the worker to a hash table called connections. This hash table stores the socket addresses and the ping counts as key, value pairs. This way, whenever *ping( )* is called, all connections in this hash table will be pinged and updated. If the ping count in the hash table is greater than 5, then the connection will be removed.

When a worker requests work, the server assigns it a section of the names using the *DataAllocator* class and adds these to another hash table called *currentNames* where the values are arrays of bytes corresponding to the names that that worker is currently processing. This way, when a worker has disconnected, the server can easily put the names back into the queue of names.

Allocating and manipulating the data set:

The server needed to be able to dynamically manipulate the dataset that we are searching from depending on how much data a client wished to volunteer in search for the correct name. A few technicalities arise that make this process a little less straight forward, first off the dataset is huge, so it is important that we make the process of parsing the names thread-safe for concurrent processing as the rest of the program runs. In our initial design we hadn't accounted for this as our test dataset was much smaller so it wasn't noticeable. The second problem is when a dataset is handed to a client but the client never finishes its work for reasons such as a disconnect or sudden power loss, so the class needs to be able to take back names for redistribution with priority ahead of all other names.

We created a *DataAllocator* class that provides the required functionality for us. The Server would have one DataAllocator per file (per dataset), in our case it's only one, and would pull names from it as Strings separated by " \n " and finally convert it to a byte array and send it to the client to work through. The DataAllocator makes use of two *ConcurrentLinkedQueue*s, one for the list of names from the file and another that holds the returned names that is then given priority over first queue. As it is concurrent it is thread safe, our *run()* function must be called before any names can be pulled from the DataAllocator as it begins parsing the names from the file and loading them into the queue using a *BufferedReader*. We had to set a maximum queue size so not to allow a memory leak or exception (as it depends on the size of the dataset), which would make the queue wait until it lessened in size to continue parsing.

The main functions in DataAllocator are the *getBytes(int noOfBytesWanted)* and *returnBytes(byte[] bytes)*. The first, *getBytes(int noOfBytesWanted)*, takes in an integer of how many bytes a client has volunteered to search through, then the function uses a *StringBuilder* to create a string of the max amount of names the client can take (or until the queue is empty), taking names from the returned names queue before the file names queue. The second, *returnBytes(byte[] bytes)*, takes in the byte array, converts it into a String array (by splitting them from " \n ") and lastly entering each name into the returned names queue. Throughout the class a variable *int namesProcessed* is kept to keep track of the number of names sent out to be processed at that point in time, fluctuating between names sent out and those returned, which can then be accessed from the getter *getNoOfNames()*.

Statistics:

Our statistics are all stored on the server in the *Statistics* class. This stores all the relevant information to do with the workers, including how long they've been working for. Whenever a worker connects to the server, *Statistics* is updated. It creates a new stopwatch to record how long the worker is active for, and creates a count of how many names that worker has searched through. This is all done with the *addWorker()*method.

Whenever a worker has finished a chunk of names in *Connection*, it will update the Statistics. It will run the *updateStats( )* method, which takes in the *ID* of the worker and the number of names that it has processed. *updateStats( )* increases the *timeSpent* variable, and the *namesSearchedTotal* variables with how many names the worker processed and how long it spent doing it. When the name is eventually found, all the stats from the current workers are added to the totals and then the totals are printed.

Packets:

In our program the only form of communication between the Server and it's Clients is through the use of packets. In our initial design we decided to be very generous with our packets, creating an individual type for almost every action, this was hard to read and needed refining to what it is now. We use a superclass *PacketContent* that all packet types extend from, each packet class is quite simple, only differing in what data they transfer and other smaller specialised requirements.

Overall we have 5 types of packets that extend *PacketContent*, *AcknowledgeSetup*, *GenericActionPacket*, *GenericTimedActionPacket*, *WorkloadPacket* and *WorkRequest*. *GenericActionPacket* and *GenericTimedActionPacket*, is a basic packet type used for a few functions in the program that don't require any processing or function calls, just an indicator to begin a new phase for the Server or Client, it sends info from one Node to another, indicating some action that must be taken by the receiver, a *Timer* is used for some incase it does not receive a response in time for it to be resent. These basic packets are used for when a worker wishes to begin working, when the server wants to ping a worker, the response ping from the worker, when a worker completes a workload and lastly when a worker wishes to terminate it's work.

The *AcknowledgeSetup* packet is in response to the *GenericActionPacket* asking to join and begin working and indicate it is prepared to accept work requests from it. A worker then sends the *WorkRequest* packet indicating how much data it would like to receive through the function *getCapacity()* which is then delivered in the form of a *WorkloadPacket.* The *PacketContent* class organises each of these subclasses by assigning types to each of its unique packets and thus creates the datagram packets and oversees them.

Server and Client:

The Server and the individual Client classes are the main classes of the program, both type of *Node.* Together they use utilise and pull together all the other classes built to give functionality and create a functioning protocol as intended from the start. Both classes are quite intricate and difficult to understand by reading the code alone from their myriad of function calls between the many packet types, the data allocator and the connection list.

The basis of Server is easy to read, there is the constructor, *onReceipt()* for when a packet is received and lastly *start()* to begin the server, yet the contents are difficult to understand and complicated without a coherent understanding of every aspect of our protocol. *start()* creates the multiple *Connection()* and starts the *DataAllocator* to process and parse the dataset. In *onReceipt()* the Server takes in a packet and depending on its type the Server decides the next sequence of flow for the program to continue to, if a new worker wishes to begin working, if an existing worker requests more data, a response ping to indicate a worker is still working and finally if the name has been found to terminate all workers.

The Client class has an identical structure to the Server, with it's constructor, *onReceipt()* and *start()* functions. The *start()* function sends the initial request to join in the search and ensures that while name hasn't been found it will continue to request for more data and process it. The *onReceipt()* takes in a packet and depending on its type decides the next sequence of flow for the Client to continue to, to begin requesting workloads, to process data just received, to return a ping from the server or to terminate work. The client uses a *DataProcessor* class to search through the data given and returns if it has been found or not. Unlike the Server, each Client keeps count of it's own statistics for show at the end of the process.

Implementation Detail:

Throughout most of our protocol the implementation of our functionality is straight forward yet for some, like the multiple connections and the data allocation, it is more complex. Here we will explain their coding in more detail with a specific look at the code.

Connection:

We had some trouble implementing our *Connection* class until we decided to use the *ConcurrentHashMap* data structure which allowed us to make it thread safe and eventually functional. We chose to use two *ConcurrentHashMap*s because of the fast run time in returning all the workers connected to the map to the server at once, this allowed us to terminate them all after the name is found in a fast and efficient manner. The first hash map *connections* uses the socket address as the key and keeps the number of pings sent to it, the second also uses the socket address as the key but holds the file names that it is currently processing.

The most complex code lies in the *ping()* function, as it uses functionality from the server, packets and its own connection functions.

```
GenericActionPacket ping;
for (SocketAddress worker : connections.keySet()) {

        ping = new GenericActionPacket(worker, server,
        PacketContent.PING_WORKER);
        ping.send();
        connections.put(worker, connections.get(worker) + 1);

        if (connections.get(worker) > thresholdPing) {
                server.returnNames(currentNames.get(worker));
                connections.remove(worker);
        }
}
```

Part of the *ping()* function in Connection.

The for loop iterates through the hash map and sends a new ping packet to each worker to ensure they're still active. If it exceeds a set number of pings (meaning it's inactive) it is removed from the protocol and its names are returned to the list to be processed.

DataAllocator:

We made our *DataAllocator* class *Runnable* to allow for the *run()* function to concurrently and continuously process names from the while alongside the rest of the program as it operates. We use a *ConcurrentLinkedQueue()* to dynamically hold the names.

```
BufferedReader buffer = new BufferedReader(new FileReader(file));
String str_line;
while ((str_line = buffer.readLine()) != null) {
        if (namesQueue.size() > maxQueueSize) {
                Thread.sleep(500);
                System.out.println("Thread sleeping");
        }
        else {
                namesQueue.add(str_line + "\n");
        }
}
buffer.close();
```

Part of the *run()* function in DataAllocator.

We needed to ensure the queue never became too larger as it would cause exceptions and slow down the process tremendously, to do this we set the simple *maxQueueSize* variable and ensured the queue never grew past this size.

The function *getBytes* uses a StringBuilder to efficiently create one large string of all the names to hand out to the worker that is under the specified number of bytes requested.

```
StringBuilder namesToSend = new StringBuilder();

// Calculates how many names should be given from the queue.
while (currentByteSize < noOfBytesWanted
                        && namesQueue.peek() != null)
{
        currentByteSize += namesQueue.peek().length();
        namesToSend.append(namesQueue.remove());
        namesProcessed++;
}
return namesToSend.toString();
```

Part of the *getBytes()* function in DataAllocator.

Here the queue is utilised fully to achieve the desired string. A StringBuilder is used instead of the primitive String type to allow us to increase efficiency. The *returnBytes()* function works in a similar manner by utilising the queue to return items into it to be processed once again.

Reflection & Conclusion:

Overall our project was a success, our protocol worked as intended and we worked together well to achieve our final program. Throughout the entire process we overcame many obstacles as a team that required us to redesign and change our project for the better. Together we learnt a lot from threading to concurrent data structures.

As to who did what, Tomas was the clear leader for this project, by coming up with the basis of the overall plan. Together we coded and implemented our separate parts as seen by the comments at the start of each class and brought them together as one. Lastly the report was written by Calvin and Rob.

If we were to begin again we would make some small changes such as readability and comments, to aid users understanding how it works. The functionality of the protocol works well and achieves what we set out to do, to implement a design for distributed applications mirroring Volunteer Computing!