

# Telecommunications II

## Assignment 1 – Sockets & Headers

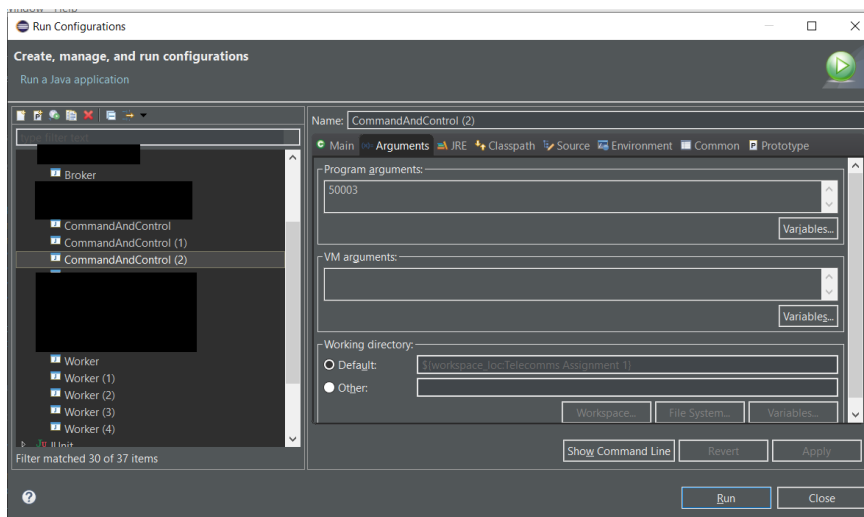
Stephen Rowe – ID 14319662

Date: 3<sup>rd</sup> November 2019

### Assignment Approach

The aim of our assignment was to create three separate applications, C&C, Broker, and Worker, which would communicate to each other by using a protocol that we designed ourselves. We also had to make use of Datagram Packets, Sockets, and Threads within each application in order to facilitate this communication.

I decided to do this assignment entirely within Eclipse, as this was the environment that I was most familiar with, and Eclipse provided all the features I needed. Each application was a class within my project folder which had its own main() method, and utilised other classes within that folder (e.g. Node, SNDContent, ACKContent, NodeData, and TimerFlowControl). In order to facilitate having multiple Workers and multiple C&Cs, I created multiple run configurations of both classes and passed the port number as the argument in each case.



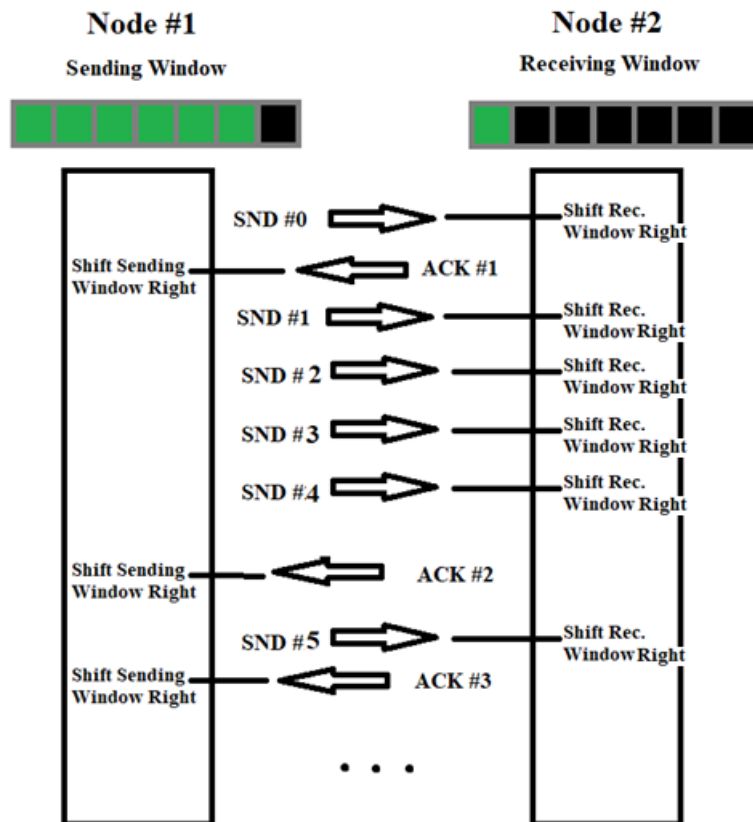
I opted to implement the Go-Back-N ARQ for Flow Control for my system, which will be elaborated on later. My system has all the following features, as outlined in the assignment instructions:

- Workers accept names as input, can volunteer and quit working
- C&C accepts work description as input, and forwards this to the Broker
- Broker maintains list of current workers and designates them work
- Worker's complete tasks and notify the Broker
- C&Cs can request multiple workers to complete a task
- Brokers, C&Cs and Workers operate based on their instructions being acknowledged by the connected node, and cannot continue until (up to) the last 15 packets have been received (due to Go-Back-N).
- Multiple workers and multiple C&Cs possible.

## Go-Back-N ARQ

Go-Back-N ARQ is a Flow Control in which packets are numbered by  $m$  bits. The node sending data has a Sending Window of  $2^m - 1$ , and the node received data has a Receiving Window of 1. In my applications, I used a window size of 15 for my Sending Windows.

The below image demonstrates Go-Back-N in a normal scenario:



Go-Back-N allows the sender to **burst-send up to 15 packets at a time, without having to wait for an ACK every time**. The node receiving the packets sends an ACK for each packet that has been successfully received and error checked. If a receiver gets an ACK, then it will shift the sending window right by 1, meaning that it will not send that packet again. If a packet is sent and lost in transit, then the **receiver node does not accept any subsequent packets that don't match the expected packet number** (i.e. the one element in the receiver window). As a result, the sender resends all the packets within its Sending Window, until it receives the relevant ACKs to move it forward. If a sender is sending a packet that has already been received, then the receiver sends them back an ACK that notifies the sender of the **next packet that the receiver is expecting**.

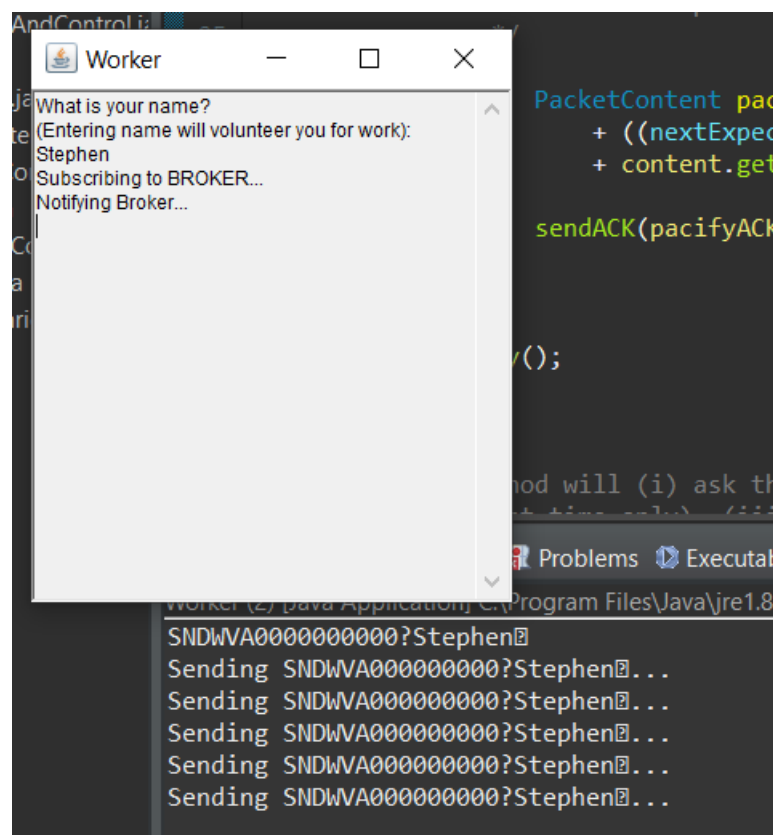
I implemented Go-Back-N within my applications by using Timer arrays and a new class TimerFlowControl. The timer array itself acted as the Go-Back-N window. It was of size 16, but was required to always have a null element, so that the window was always of size 15. When an information packet was to be sent, a Timer was created for it, that would operate a TimerFlowControl object every time the timer ran out. My TimerFlowControl object accepted the host socket (i.e. sender) and the actual packet (with receiver socket already set) as parameters, and would send the packet. The timer would continue to send the packet until the relevant ACK was received, in which case the Timer would be cancelled, the element on the array would be nullified, and the next packet in the window would be allowed to be sent.

As all of our nodes (Worker, C&C, Broker) would be both sending and receiving information packets, then this meant that **each node would have a Sending Window and a Receiving Window**, which had to be iterated cyclically (e.g. 13, 14, 15, 1, 2, 3...).

While sending information packets would be repeated until receiving the relevant ACK, we have to treat sending ACKs differently. When an ACK is sent from a node, **it is only sent once, and therefore does not join our Sending Window**. However, in the case that an ACK that we sent is lost, we have to send a “pacifying” ACK, that notifies the sender that we have received their packet, and what the next expected packet (i.e. the element on our receiving window) is.

When accepting ACKs, I implemented **the ability to ACK multiple packets at once**. This means that, if packets 1, 2, 3, 4 were being sent and received but the ACKs for each were lost, then the receiver can send back a single ACK with packet number 5, meaning that it is expecting the packet number of 5. The sender of the packets then goes through each element in the Go-Back-N Sending Window, and accepts its instructions as if it got an ACK for all 4 packets. Please see the `acceptACK()` methods in each of the Broker, C&C, and Worker classes to see how this is done. We have to **iterate backwards through the Sending Window in order to process all packets as being accepted**.

To best demonstrate Go-Back-N in my program, I will run a Worker Application without a Broker.



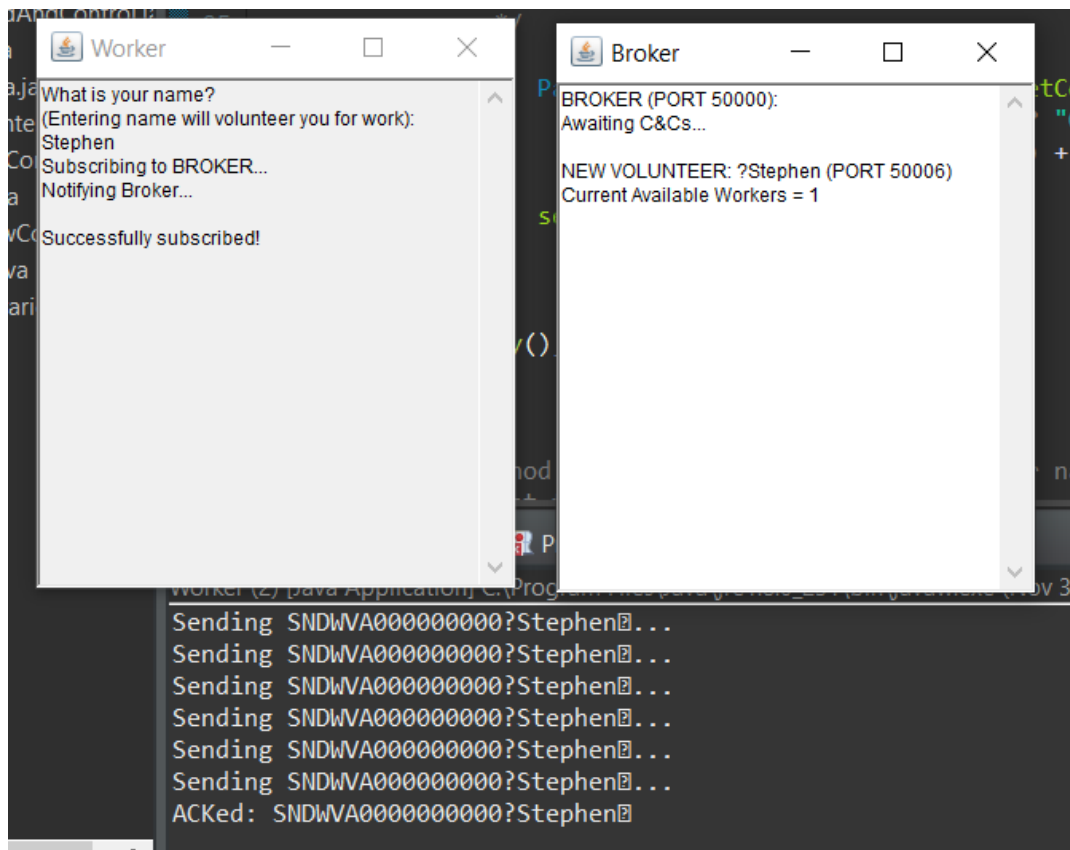
The screenshot shows an IDE with a window titled "Worker". The window has a text input field with the prompt "What is your name?" and a button labeled "(Entering name will volunteer you for work):". Below the input field, the text "Stephen" is entered. The window also displays "Subscribing to BROKER..." and "Notifying Broker...". In the background, a code editor shows a snippet of Java code: 

```
PacketContent pac
+ ((nextExpec
+ content.get
sendACK(pacifyACK
();
```

 The console at the bottom shows the following output: 

```
Worker (2) java Application C:\Program Files\Java\jre1.8.
SNDWVA000000000?Stephen
Sending SNDWVA000000000?Stephen...
Sending SNDWVA000000000?Stephen...
Sending SNDWVA000000000?Stephen...
Sending SNDWVA000000000?Stephen...
Sending SNDWVA000000000?Stephen...
```

As you can see in the console, the Worker is repeatedly trying to send a packet to the Broker, but the broker is not active. If we now open up a Broker *without* restarting the Worker:



The message is accepted by the Broker and the Worker has received an ACK, meaning that this packet has been removed from the Go-Back-N window, and will no longer be sent. **This would also work the same if the Worker was sending up to 15 packets to the Broker**, it would simply wait until it got an ACK back from the Broker before proceeding.

## Packet Headers & Protocols

Please see the class `SNDContent` to see how the packet headers are used in practice.

My packet headers consisted of the following:

- Packet Type - SND and ACK
- Content Type - JOB, WVA/WVU, CRI, CMP
- Packet Number - ##
- Originating C&C Number - @@ (Up to 99 possible C&Cs)
- No. Workers Given Job - %%
- Job ID - &&&&

Each packet that is sent is constructed with a header in the above format. SND denotes that the packet is an information packet, ACK denotes that it is an ACK (an `ACKContent` class was created specifically for ACKs, will discuss later). Content types were necessary to convey what kind of packet this was, and how it should be processed by the receiving node (JOB is a new job listing, WVA/WVU signifies that the worker wants to subscribe or unsubscribe from the Broker, CRI means that there is a new C&C available). Originating C&C number is used by all three classes to keep track of the ID Number of the C&C that requested the job. This ID number is designated by the Broker, and depends on how many C&Cs are currently present.

No. Workers Given Job denotes how many times the task needs to be done, as requested by the C&C. When a C&C requests a number of workers to do a job and there are enough workers for this, then this value is accepted. However, if the C&C requests all workers to do the job, or more workers than are available, then the

Broker must tell the C&C how many workers are doing the job (by sending it back a JOB packet with “No. Workers Given Job” set to the total number of workers), and now both the Broker and the C&C knows how many different tasks must be completed before they can each remove that particular job from its job lists.

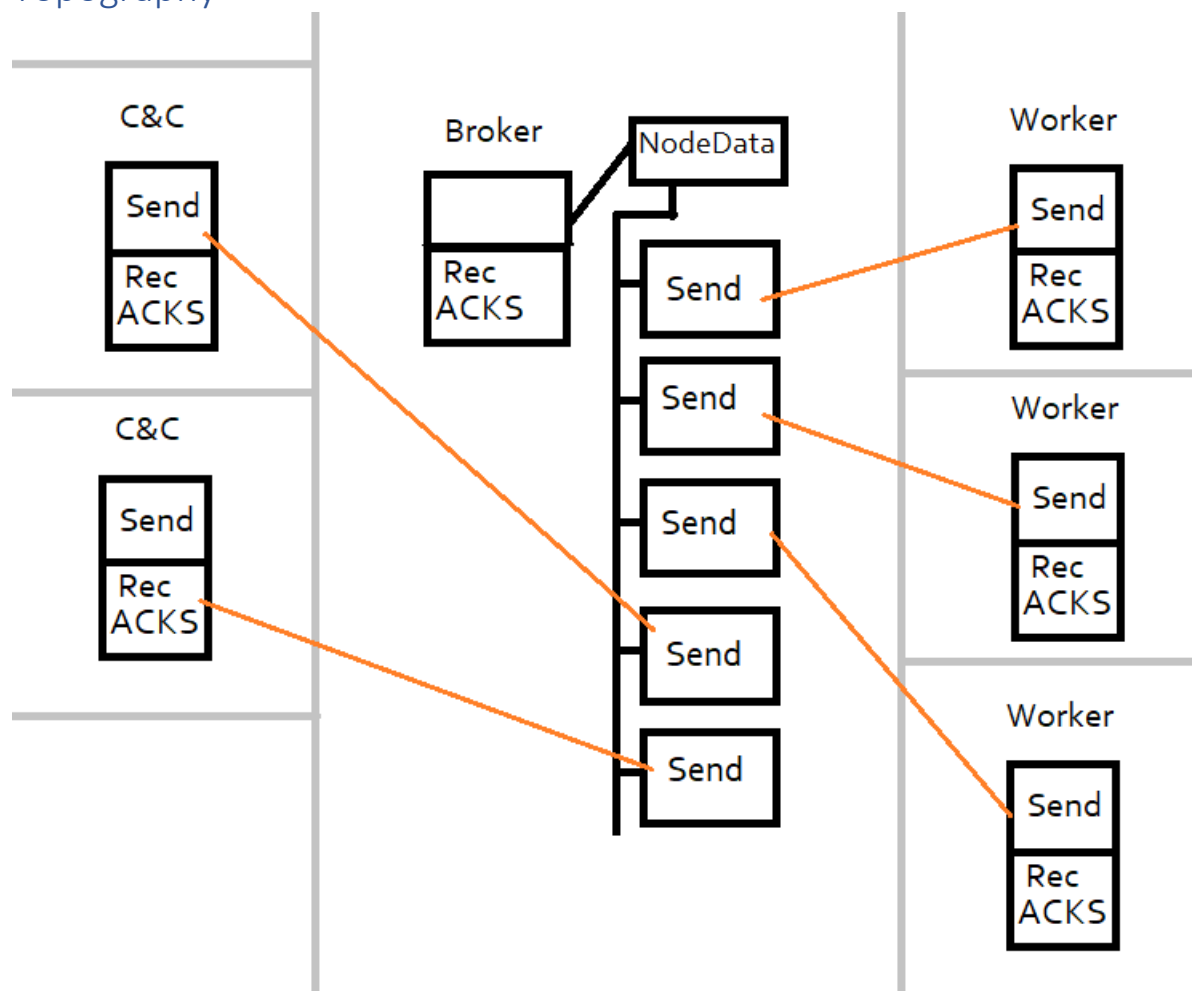
Job ID is simply a number that the C&C produces so that the Broker can more easily recognise the relevant job while it is going from Broker to Worker and Broker to C&C.

I originally wrote my applications with just a PacketContent class, that would be used for both ACKs and SNDs. However, I realised that ACKs consisted of far too much unnecessary information (as they would consist of all of the header content above, just lacking the actual packet string). There was also the factor that, if a packet was sent from a Broker to a Worker, all the ACK would be used for would be getting its packet number, and not much else, since the process of accepting ACKs involved iterating backwards through the senders Go-Back-N windows and processing these instructions internally. I decided to separate the SNDs and ACKs into two different classes. Now, every time a packet is received, an ACKContent is created, which can return a validACK Boolean, which lets us know if we should create a SNDContent from the delivered packet instead.

I also realize that most of this header content could have been denoted with single bits as opposed to strings (such as ACK and SND), however I found developing much easier when I used the strings, so I stuck with it.

When a Datagram Packet is received and it is *not* an ACK, it's converted into a SNDContent, which will categorize all the elements of the packet for easy access by our nodes. Each node will then operate on this instruction, depending on these values. This best summarises our protocol.

## Topography



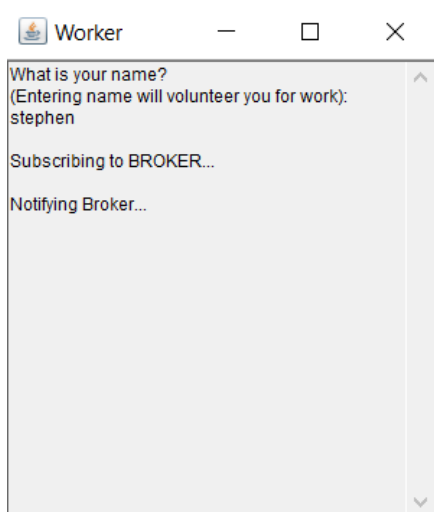
Both a C&C application and a Worker application are able to keep track of their own go-Back-N windows, as they are each only connected to one node, the Broker. However, the broker must keep track of the go-Back-N window for each node. This is done through the NodeData class. This keeps track of the next packet number the Broker should send to the Node, and the next packet the Broker should expect from the node. In the case of a NodeData object being a worker, the broker also keeps track of the jobs that have been designated to that worker.

## Worker

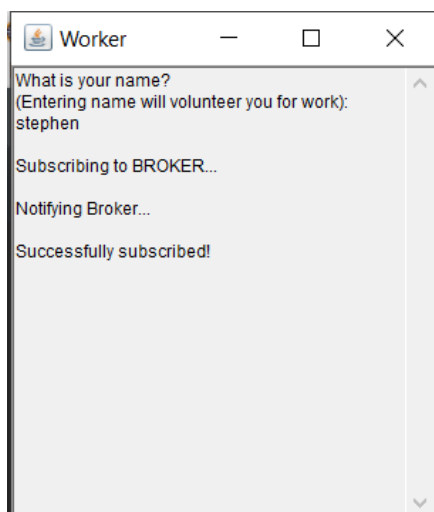
Of the three types of applications, the Worker was the easiest to design, although all three had the same fundamental structures.

The only two sockets that the worker needs to keep track of were its own ("*workerSRCPort*") and the Broker's ("*dstAddress*" was set by creating a new *InetSocketAddress* with "*localhost*" and the Broker's port). It also needed to keep track of the jobs it has been designated (*currentJobs*), and if it is subscribed to the Broker (through the Booleans *availableForWork* and *currentlyChangingSubscription*).

When the Worker application starts up (after being assigned a source port by the arguments of the program), the program asks the user to input a name. Once a name has been inputted, the method *changeAvailability()* is called. In this case, this method will send the Broker a "WVA" packet, accompanied with the worker's name, so that the Broker recognizes the worker and knows that it can send the worker jobs.



Once the Broker sends an ACK back, the worker knows to start expecting jobs:



For actually carrying out jobs (without requiring any input from the user), I created a Timer and a Timer-Task class, called doJobTimer and doJobClass respectively.

The Timer would go off every 10-15 seconds (this is different for each worker as the number of seconds is randomly generated between these values). Once the timer goes off, the class doJobClass is called, which would only call the worker class doAJob() (there is probably a better way of doing this, but was the quickest way of implementing what I wanted from the timer).

If the method doAJob() was called, there were two possibilities:

- Worker quits
- Worker randomly selects a job and marks it as complete

The likelihood of the worker quitting at any given call depends on the following lines:

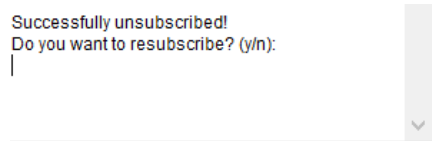
```
if(new Random().nextInt(30) == 15)
    changeAvailability();
```

In the above case, there is a 1 in 30 chance that the worker will quit this time. I wanted next to no actual user input in the worker class after the user gave their name, as any other input would impede the flow of packets due to the nature of the synchronized threads (will elaborate on this point further in the C&C description), so this is why I chose for the worker to both unsubscribe and do a job randomly.

```
else if(currentJobs.size() > 0)
{
    SNDContent jobChosen = currentJobs.get(new Random().nextInt(currentJobs.size()));
    SNDContent jobCompleted = new SNDContent(jobChosen.toString());
    jobCompleted.resetContentType("CMP");
    sendPacket(jobCompleted);
    terminal.print("\nJOB COMPLETE: " + jobChosen.getPacketContent() + " (Job ID " + jobChosen.getJobID());
    currentJobs.remove(jobChosen);
}
```

In this case, the Worker will not unsubscribe, but will instead do a job. In the above lines, a random job is chosen from the worker's list of assigned jobs. The packet is taken, the content type is reset to CMP (complete), and this is sent to the Broker through the sendPacket() method.

If the Worker does decide to unsubscribe, then the user will be asked if they want to re-subscribe, in which case the process of notifying the Broker is repeated, without having to ask the Worker for their name again.



```
Successfully unsubscribed!
Do you want to resubscribe? (y/n):
|
```

In the method changeAvailability(), the worker will either subscribe or un-subscribe from the Broker, depending on its current status (using the availableForWork boolean).

```
private static void changeAvailability() {
    SNDContent availabilityNotification = null;
    if (!availableForWork) // if subscribing
    {
        terminal.println("\nSubscribing to BROKER...");
        availabilityNotification = new SNDContent("SNDwVA0000000000" + workerName + '\u0003');
    }
    else if (availableForWork) // if unsubscribing
    {
        terminal.println("\nUnsubscribing from BROKER...");
        availabilityNotification = new SNDContent("SNDwVU0000000000" + workerName + '\u0003');
        currentJobs.clear();
    }

    terminal.println("\nNotifying Broker...");
    currentlyChangingSubscription = true;
    availableForWork = !availableForWork;
    sendPacket(availabilityNotification);
}
```

If they unsubscribe, then their list of jobs is immediately cleared, so that no more jobs are done in the time frame between unsubscribing and the Broker acknowledging the un-subscription. Another boolean, `currentlyChangingSubscription`, also prevents any unintended consequences in this same time-frame.

I will explain the `sendPacket()` and `AcceptACKs()` method here, but they will be the exact same in the C&C class and very similar in the Broker class. The `sendPacket()` method is as follows:

```
private static void sendPacket(SNDContent packetContent){
    // One element on the goBackN window array must always be null.
    while(goBackNWindowSize >= 15)
    {
        try { Thread.sleep(500);
        } catch (InterruptedException e) { e.printStackTrace(); }
    }

    // reset the PacketNumber of the Packet so it matches up with the
    // Broker's next expected Packet
    packetContent.resetPacketNumber(nextSentPackNum);

    // turn into DatagramPacket and set destination for packet
    DatagramPacket packetToSend = packetContent.toDatagramPacket();
    packetToSend.setSocketAddress(dstAddress);

    // start new timer for Go-Back-N
    Timer packetTimer = new Timer();
    TimerFlowControl ARQ = new TimerFlowControl(socket, packetToSend);
    packetTimer.schedule(ARQ, 0, 3000);

    // Make 2 arrays where any given i on both arrays are for the
    // matching PacketContent and timer.
    goBackNWindow[nextSentPackNum] = packetTimer;
    goBackNWindowContent[nextSentPackNum] = packetContent;
    goBackNWindowSize++;

    //iterate nextSentPackNum
    nextSentPackNum = (nextSentPackNum + 1) % 16;
}
```

First, if `goBackNWindowSize >= 15`, then the thread sleeps. What this means is, we have a limit in the Go-Back-N window of 15, and if the Worker is currently trying to send 15 packets that have yet to be ACKed, then we must wait until an ACK returns.

After this, any packet that we chose to send must be modified so that the packet number of the packet matches the packet number of our Go-Back-N window. This is done by calling the `resetPacketNumber()` method in the `SNDContent` class. After creating a `Datagram Packet` and setting the socket address to the Broker's address, we create a new `Timer` and `TimerFlowControl` for the packet. The timer will call the `TimerFlowControl` every 3 seconds, resending the packet every time. An array for this will allow us to access and cancel this timer at a later point (i.e. when the packet is ACKed). Finally, we must increment `nextSentPackNum`, but do so in a way that once it reaches 15, it goes back down to 0, and repeats this process.

The most important aspect of the `acceptACKs()` method is cancelling the timers of outgoing packets. The following pseudocode explains:



```

acceptACKs(int latestACK){

    latestACK--;

    packetTimer = goBackNWindow[latestACK];
    packetContent = goBackNWindow[latestACK];

    while(packetIteration != null)
    {
        packetTimer.cancelTimer();
        goBackNWindow[latestACK] = null;

        if(packetContent == WVA)
            print(Successfully Subscribed!)
        else if(packetContent == WVU
        {
            nullifyAllGoBackNWindowElements();
            cancelAllGoBackNWindowTimers();
            nextSentPackNum = 0;
            print(Successfully subscribed!);
        }
        else if(packetContent == CMP)
        {
            print(Broker has accepted our work for this job!);
        }

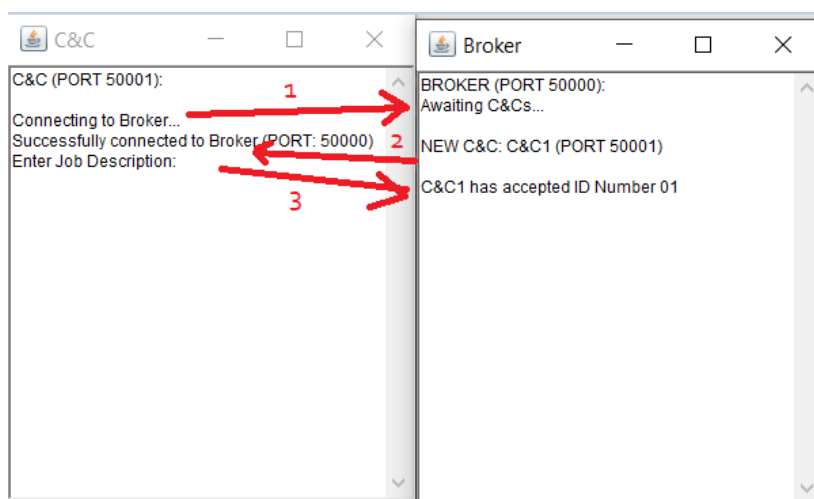
        latestACK--;
        packetTimer = goBackNWindow[latestACK];
        packetContent = goBackNWindow[latestACK];
    }
}

```

Since the ACK received will be the next packet that the receiver expects, then we must decrement it by one first, then go through each element *before* this packet number and treat all the packets as ACKed, nullifying and cancelling their timers.

## C&C

The C&C was closely modelled on the Worker, with a number of significant differences. Like the Worker, the C&C must be acknowledge by the Broker. However, unlike the Worker, the C&C must be assigned an ID that it must append to each job, so that the job is returned to the correct C&C. This is done by immediately sending a "CRI" (C&C Request ID) packet to the Broker. The Broker then registers the C&C as a connected node. The Broker then ACKs this packet, and follows it up with "CRI" packet of its own, this time containing the ID of the Broker, which the C&C then accepts.



Job lists, Timers, Go-Back-N Windows, sending Packets, and dealing with ACKs all works fundamentally the same for the C&C as it does for the worker, so there is no need to reiterate these points.

When the start() method of the C&C is called, it asks the user to input the name of a job, and then how many workers they want to do that job. The fact that the user is constantly asked for input is the biggest flaw in my application, as this severely impedes the packet receiving processes. Since there is synchronization between the onReceipt() and start() methods, i.e. they take turns to be carried out, then if we are waiting for the user to input, none of the packets being sent to the C&C are being received. Once onReceipt() is called, then it can only be called once before start() runs again, meaning that only one packet is registered. I experimented with the *synchronized* status, including removing the term *synchronized* altogether, but I could not get the desired result. Regardless, I can still demonstrate all the features of my program without this, so it's not a huge issue, but it's worth noting.

When the user decides to create a job and inputs the relevant data, the method createJob() is called:

```
private SNDContent createNewJob(String jobContent, String numWorkers)
{
    // When the C&C wants to create a new job, we just need the String of the job content,
    // and the number of workers they want to do this job. From there, we can construct
    // a PacketContent with this information.
    int number = Integer.parseInt(numWorkers);

    if(number < 10 && number >= 0)
        numWorkers = "0" + numWorkers;
    SNDContent newJob = new SNDContent("SNDJOB00" + idNumber + numWorkers
                                       + getJobIDToString() + jobContent);

    jobID++;
    if(jobID > 9999) jobID = 0;

    return newJob;
}
```

All this method essentially does is create a SNDContent object with all the relevant values set (such as the C&Cs ID number, how many workers they are requesting, and the name of the job. jobID is incremented from zero.

If the Broker sends either a JOB (i.e. telling the C&C how many workers are actually working on a job) or a CMP (i.e. worker has completed a job) packet, then we have to find the relevant job in our list of jobs. This is done with a for-loop with the following condition:

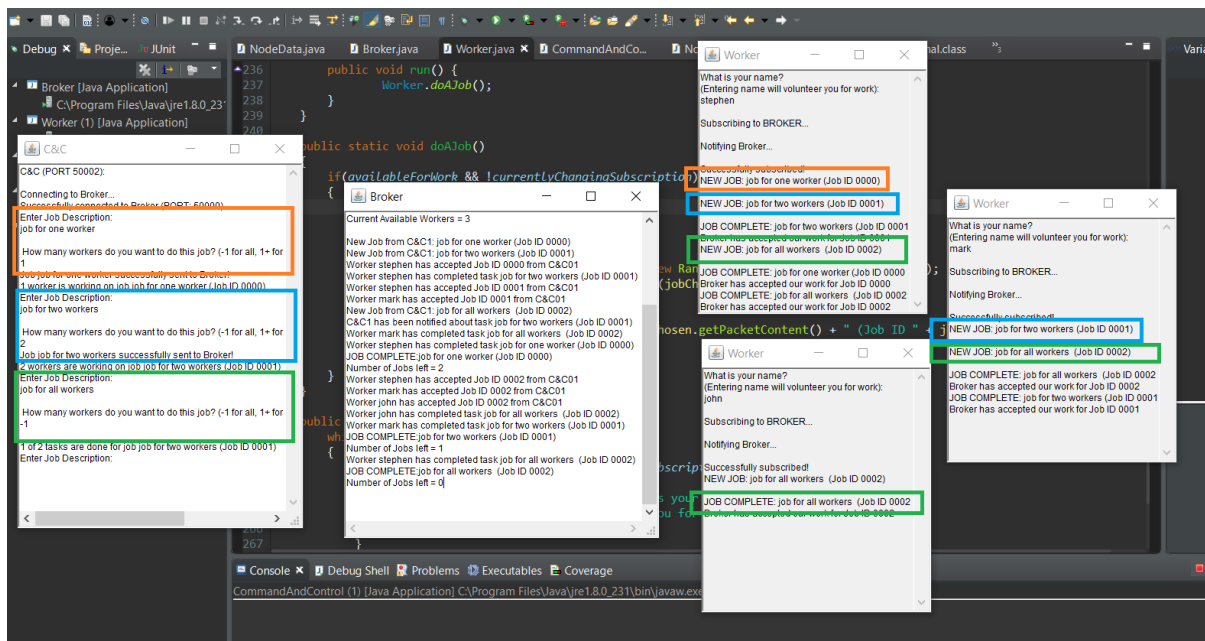
```
if (iterationJob.getOriginatingCAndC().equals(content.getOriginatingCAndC())
    && iterationJob.getJobID().equals(content.getJobID()))
```

This means that all we have to do is match the C&C ID and the Job ID of the packet with what we already have in our ArrayList in order to accurately identify which job the Broker is talking about.

Below we can see how the C&C and Broker interact, from the terminal's point of view:



Here is a demonstration of choosing how many workers do a specific job:



## Broker

The Broker was the most difficult class to write as there was much more for the Broker to keep track of than either of the two previous classes. Since this application would be most central to my system, I started working on this first, but continuously had to update it as the other two classes were being developed.

Broker overall has a similar structure to both Worker and C&C, but there is one significant difference – it must connect to multiple nodes, as opposed to just one, and as a result requires a system for considering the Go-Back-N flow control for each node. In order to do this, I created a NodeData class.

If a Worker or a C&C requested to be recognized by the Broker (i.e. with a WVA or CRI packet, respectively), but the Broker has not connected to this node before, a NodeData object is created, and added to our array list of *connectedWorkers/connectedCAndCs*. This NodeData class carries out the SendPacket and Go-Back-N Window functionality of each node. The exception is made for dealing with ACKs, and receiving SND instructions from other nodes, as shown in my topography diagram. The Broker class itself deals with ACKs, as we have to adjust the Broker's array lists and terminal outputs, which are inaccessible from within the NodeData class. We also follow worker and C&C instructions with the methods *processWorkerInstruction()* and *processCAndCInstruction()* respectively.

Registering a new worker node is just a matter of adding the worker to our connectedWorker array list and sending an ACK back. Register a C&C involves adding the C&C to our connectedCAndCs array list, sending an ACK back, but then sending the C&C another CRI packet with its designated ID. This ID is determined by the number of C&Cs already present, so its just an increment of one.

The acceptACKs() is essentially the same as with the C&C and Worker classes, except that it must have a NodeData *nodeDeliveredFrom* parameter, which was determined within the findNode() method, which iterates through both the connectedWorkers and connectedCAndCs array lists to find a matching port number. Once this is established, then we access the nodeDeliveredFrom's Go-Back-N arrays from the Broker class, and modify them as needed.

The below pseudocode demonstrates how the Broker deals with a worker notifying them that a job is done, and dealing with a worker unsubscribing, subsequently redistributing that worker's jobs:

```

processWorkerInstruction(nodeDeliveredFrom, SNDContent packet)
{
    if(packet = CMP)
    {
        NodeData relevantCAndC = findRelevantCAndC();
        SNDContent relevantJob = findRelevantJobFromBrokerAllJobList();

        relevantJob.markOneTaskDone(); //i.e. decrement numTasksUntilJobComplete
                                         // variable in SNDContent

        if(relevantJob.numTasksUntilJobComplete == 0)
            allJobs.remove(relevantJob);

        SNDContent sendTaskCompleteToCAndC = new SNDContent();
        relevantCAndC.sendPacket(sendTaskCompleteToCAndC);

        ACKContent completedJobACKToWorker = new ACKContent();
        sendACK(nodeDeliveredFrom, completedJobACKToWorker);

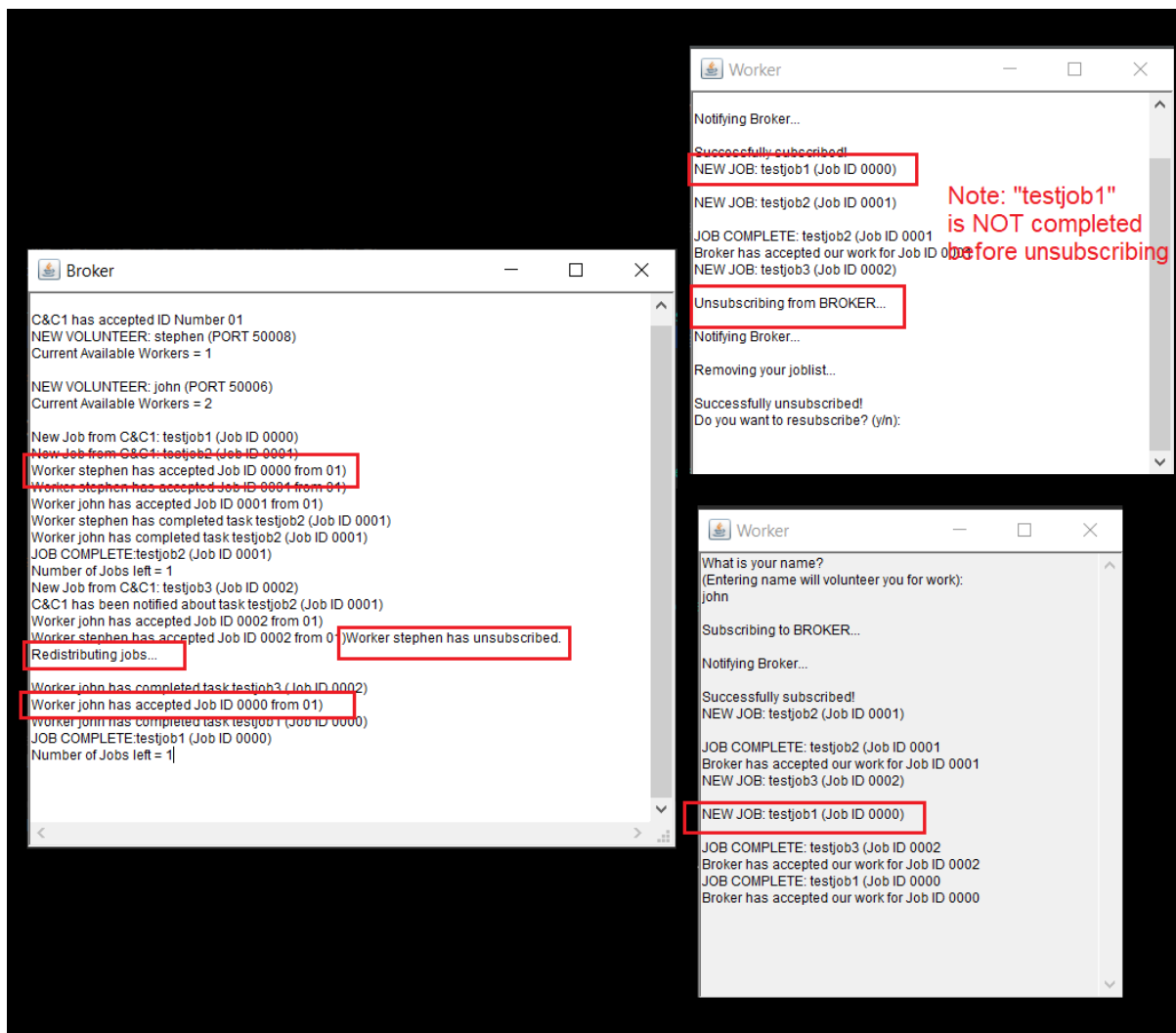
    }
    if(packet = WVU)
    {
        connectedWorkers.remove(nodeDeliveredFrom);

        for(int i = 0; i < nodeDeliveredFrom.designatedJobs.size(); i++)
        {
            jobsPendingDistribution.add(designatedJobs.get(i));
        }

        ACKContent unsubscribeACKToWorker = new ACKContent();
        sendACK(nodeDeliveredFrom, unsubscribeACKToWorker);
    }
}

```

Whenever a Broker received a job from a C&C, this job is added to two arrays, *allJobs* and *jobsPendingDistribution*. Every time the Broker's start() method continues, each element of *jobsPendingDistribution* is distributed to workers a number of times (depending on the value of *numWorkersForJob*), but no worker is sent the same job twice within this loop. Once all jobs have been distributed within this loop, then *jobsPendingDistribution* is emptied. Whenever a worker unsubscribes, as you can see from the above pseudocode, the job is put into *jobsPendingDistribution* again. Below is a demonstration of a job being redistributed after the worker unsubscribes:



In the above example, worker “John” has received a job that “Stephen” didn’t do. The following example also works:

Let’s say a C&C requests all workers to do a job, “write lab report”. The Broker currently has 4 workers, so the Broker tells the C&C that 4 workers are working on this job. The job is successfully distributed to all 4 workers. However, one worker unsubscribes. The job is then redistributed, and one of the workers now has to write two lab reports, as the C&C is expecting 4 in total.

Both of these examples work as intended. It doesn’t matter if one given worker is doing a job twice, as long as the number of tasks that the Broker and C&C agreed upon are carried out.

## Reflection

I dedicated a lot of time to this assignment and found that I got a far better understanding of threads, sockets, headers and protocols by doing this assignment than I would have by just studying a book. Actually designing headers and protocols allowed me to see the logic required to construct these features and how much or how little should be included. The use of threads and sockets has greatly improved my Java skills too. This assignment took me 30 hours.