CS3102 File Distribution Practical

140011146

2017-02-22

1 Introduction

In this practical, we were asked to design and implement a file distribution protocol that can reliably and efficiently transfer a file to multiple nodes. My protocol uses UDP multicast to transfer the file and TCP for control packets. It is a `NACK` based system where the clients send `NACK`s if they miss any packets which the server then resends.

My implementation is written in C. To run the code, please refer to the attached README file.

2 Design

For my multicast protocol, UDP is used for multicasting and TCP is used for control messages. TCP must be used for control because the control packets must move reliably between client and server and be guaranteed to arrive.

In my first implementation, I sent the entire file across with multicast before waiting for `NACK`s and resending packets that were missed. However I found a few issues with this approach:

- The size of each `NACK` varied and so there cannot be a fixed sized `NACK` packet. This made it difficult to read the `NACK` packet. If it was set to fixed sized, the server still will not know how many `NACK`s may come.
- It is difficult to checksum chunks of the file if the missing packets could be anywhere and there is no system for when the server can send which chunks of the file to checksum at a time. This means the written file on the client is only properly checked at the very end, which would make it very costly to resend the whole file if that checksum failed.

For the first issue, the `NACK`s were set to fixed sizes and the server would deal with each packet as they came, resending missing data packets as soon as a `NACK` arrived. However there was another issue where the client was still busy sending `NACK`s when the server was already resending the data packets because there were so many missing packets (due to a busy network or large filesize). The second issue was also a problem as it could not guarantee that the file written on the client's end was identical to the one sent by the server until the very end. It was costly as I found frequently that the file written was different from the file sent, requiring the entire file to be transferred again. If this failed multiple times, then the cost of resending grows greatly.

Because of these issues, I decided to change the approach and send the data packets in multiple windows (chunks), allowing a number of data packets to be sent before each `NACK`. This way, every window is checked and ensured to be correct before moving on to the next window. This way if a client has problems with the written file, only the whole window has to be sent again, not the whole file. `NACK`s can also be a fixed size as the number of missing packets per window cannot be greater than the number of packets in a window. This makes the `NACK` packet slightly inefficient as it is often has lots of empty space being sent across if the number of missing packets is much less than the window size, but this simplifies any implementation of the protocol.

The first implementation of these windows had the server not waiting for any acknowledgements from the client. If after a certain time-out the server receives no further `NACK`s, then it proceeds to the next window. This was to minimise the amount of TCP control packets that would be needed.

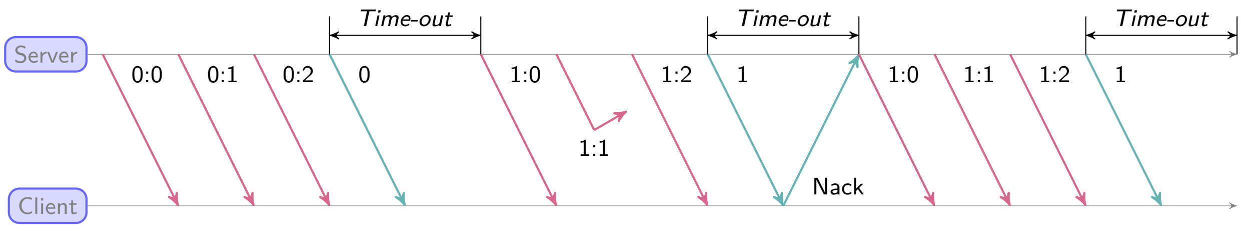

Figure 1: Flow control with time-out before next window

However when testing this implementation, there was a problem where if a client was very busy, it could be dropped due to not sending its `NACK` on time. The time-out also meant that on each window, the server must wait the time-out before it can start the next window. The length of the time-out is then very important because a shorter time-out would lead to faster transfer as there is less idle waiting per window, but would also lead to a higher chance of clients being dropped because they did not respond in time.

Because of this, I changed the implementation to send `ACK`s as well, so the server does not need to wait for any time-outs, it simply waits until the number of `ACK`s it has received is equal to the number of connected clients before sending the next window. The advantage of this change is that no clients can be dropped as the server always wait for all clients to respond and could potentially be faster than the time-out if all clients respond in a timely manner. The major disadvantage is that if a client disconnects or is blocked for a long time, the server will also block until that client becomes unblocked.

3 Protocol

3.1 Packets

3.1.1 UDP Packets

There is only one UDP packet which is the data packet. It contains the packet number which is the sequence number in the current window, the number of bytes of in its body, the current window number and the actual body (payload) of the packet.

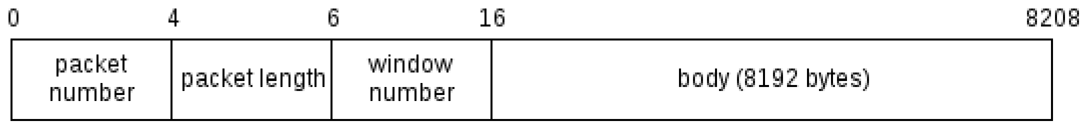

Figure 2: UDP data packet

3.1.2 TCP Packets

There are three TCP packets that are used in this protocol.

Header packet: This packet contains all the file and transfer information such as the file size, filename and packet count. It is sent at the very beginning of the transfer to every client and is only sent once.

Control packet: This packet is used by both the client and server to communicate the flow of control. There are four types of control packet: `WINDONE_MSG`, `RESEND_MSG`, `NACK_MSG` and `ACK_MSG`.

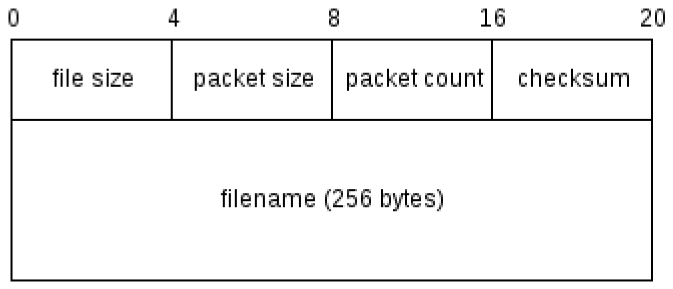

Figure 3: TCP header packet

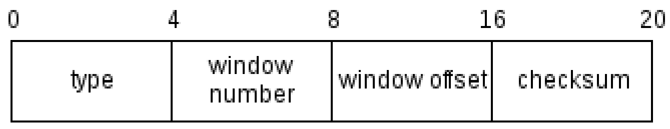

Figure 4: TCP control packet

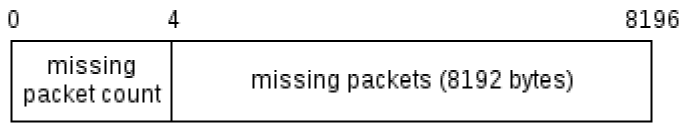

Figure 5: TCP `NACK` packet

`NACK` packet: This is the packet the client sends to the server to tell it which packets it missed in the window.

3.2 Flow control

The protocol uses the TCP control packets to determine the flow of control. Both the server and client send control packets for the end of each window and for dealing with `NACK`s and any resends that need to happen.

For `NACK`s, a time-out must be used for the client to resend the `NACK` in case the resent data packets from the server were dropped on the way. The use of the time-out could be avoided by using yet more TCP control packets to signal the end of the resend, but as the protocol already uses many different TCP control packets I didn't want to add any more to try and keep it as simple as possible.

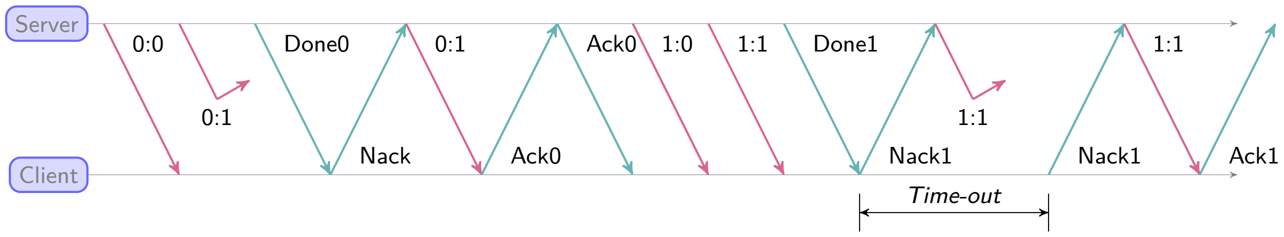

Figure 6: Time-out on sending `NACK`s on the client side

The control messages used when resending windows are necessary to make sure all the clients are kept in sync with each other. This is a final control message from the server that definitively tells all clients that the current window is over and whether the next window is a resend or actually the next window.

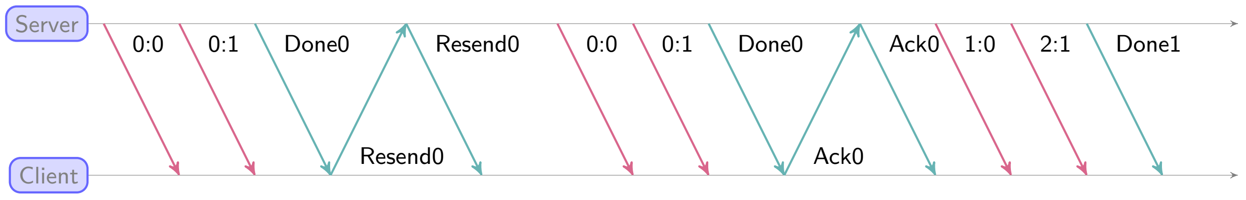

Figure 7: Flow of control for resending a window

3.3 Details

The protocol follows much of what has been explained in Section 2. A server is set up and waits for the specified number of clients to connect before starting to transfer the file to all of them. The file is broken down into windows.

Each window contains 256 packets and each packet contains 8192 bytes of data, therefore each window sends around 2Mb of data. The size of the window and packet body were gotten from stress testing sending data to clients. The larger the packet size, the better because we have to send less packets, but the risk is each packet loss is more significant. In the lab environment the chance of losing packets is quite low so I was able to choose a larger packet size. I also found that if the packet size was too large, it would take a long time to send each packet which is why I chose a large but not too large of a body size for each data packet. The window size was similarly chosen to reduce the overhead of TCP control packet exchanges at the end of every window. The window size also cannot be too large as then it would be increasingly more costly to resend the entire window again.

When the packets are sent, each is numbered from 0 to 255. At the end of each window, the server sends a control packet with message type `WINDONE_MSG` to tell all the clients it has finished transmitting that particular window. This message will also contain the window number, the checksum of the window and the offset of the file that this window has sent up to. Each client then sends a control packet back to the server with its type being either a `NACK_MSG`, `ACK_MSG` or `RESEND_MSG` depending on whether the client has missed any packets or not.

`ACK_MSG`: This is only sent if the client has received every packet and its calculated checksum is the same as the checksum provided by the server's `WINDONE_MSG`.

`RESEND_MSG`: This is sent if the client received every packet in the window, but their calculated checksum does not equal the checksum provided by the server.

`NACK_MSG`: If the client has missed any packets, it checks which packets it is missing from the window and sends a control packet of type `NACK_MSG`. After this control message, the client can then send the actual `NACK` packet which contains a list of which packets the client is missing and how many packets are missing. The server then deals with every `NACK` as they arrive. Clients can send another `NACK` after a certain time-out in case the resent packets were lost as shown in Figure 6.

While clients are sending `NACK`s, the server will continue to deal with them until all clients have sent either an `ACK_MSG` or `RESEND_MSG`. Then the server will send a final `ACK_MSG` or `RESEND_MSG` to all connected clients as shown in Figure 7. This final control message is to keep all clients in sync if some need the window to be resent but others do not.

There is a small issue I've come across where the TCP control packet for the end of the window arrived at the clients before the data packets did despite the data packets not being dropped, but simply arriving later because the network was busy. Then the client would assume it lost the last few packets of the window and send a `NACK` for them even though they turn up later. A solution to this would be to wait a small time-out on the server side before sending the control packet to the client. However because this is overhead of waiting for this time-out on every window, I decided against it. I found in most runs that very few packets (if any) packets were dropped on every window, so this time-out would make the protocol slower not faster in the lab network where it is not very busy. In a different busier network a short time-out may be a good approach if packets are being consistently dropped on every window.

4 Outcome

The data for the numbers in scp are from running the script `scp.sh` to get 100 trials and get the average (mean) time taken.

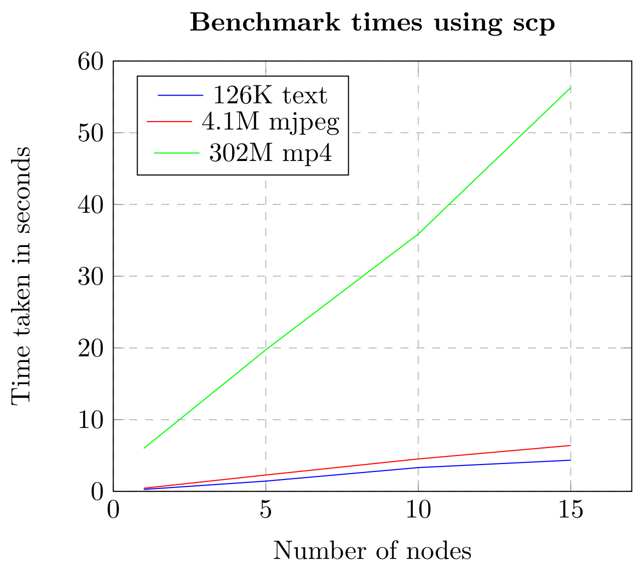

Figure 8: Time taken to transfer files with `scp`

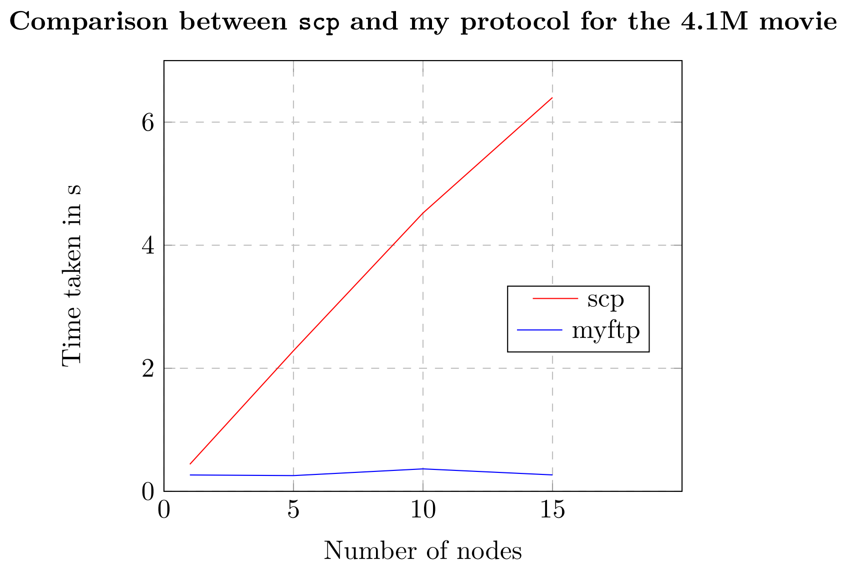

Figure 9: Time taken to transfer files with my protocol

From the benchmarked times using `scp`, we can see that there is a linear growth in time taken as the number of nodes increase. This is expected as each additional node adds around the same constant time for the file to be transferred. However, as shown in Figure 9, not only is my protocol significantly faster for the 4.1Mb movie, it also does not experience the same growth in time, making it very scalable.

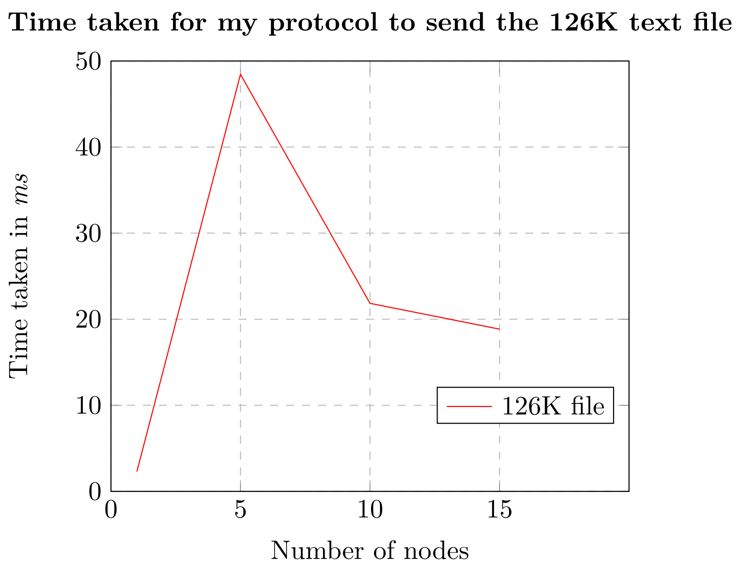

Figure 10: Time taken to transfer 126K text file with my protocol

Notice that the time taken for my protocol to send the file to 5 nodes took longer than sending it to 10 and 15 nodes as shown in Figure 10. Similarly it took longer to send the 4.1M movie to 10 nodes than it took for 15 nodes. This is because of the anomalies that are in the data. I left the anomalies when calculating the average because I was interested in whether they affected the results in some sort of pattern. The anomalies are a result of either the node being busy while sending the file or the network being busy, both of which leads to packet loss and the need to resend windows or packets. As I was gathering my results, I thought that as the number of nodes increased, the chance for these anomalies to happen should also increase, so the average time taken to send the file to more nodes would taken longer. Although my results don't support this hypothesis, I think that if I had more time to do more trials (>1000 trials), then this pattern would be more apparent.

Nodes	1	5	10	15
Average time taken in ms	14504	15719	19245	20346

Figure 11: Time to send 302M mp4 with my protocol

Having to send more packets also seems to affect this result as shown by the data for the 302M mp4 in Figure 11. Because this file is much larger, the anomalies are more common as more packets are likely to be dropped. Here we can see more nodes taking longer though if we look at each individual trial, there are still trials where the sending to 15 nodes took a similar amount of time as sending to 1 or 5 nodes.

All data can be found in the file results.ods or by runing the scripts `scp.sh`.

5 Conclusion

I have designed and implemented a file distribution protocol in C that is reliable and fast. I know my protocol is reliable as I have tested it with `netem`, emulating a percentage loss of packets and the file can still be sent and received properly. Apart from the checksum, I also `diff` the received file stored locally and the networked file to make sure it is identical. It is also scalable as having more nodes does not result in a linear increase in time the way sequential `scp` would. From testing my protocol, I have found that the more nodes and packets in the transmission, the more time it may take due to an increased chance of dropping packets. Although more scalable than sequential copying of the file, using my protocol for many nodes (>100 nodes) maybe result in a very busy network which drops a significant amount of packets and therefore still takes increased time for increased number of nodes.