CS3102 Practical 1 : A Scalable, Efficient and Reliable File Distribution Protocol

150000845

February 2017

1 Introduction

The aim of this practical is to produce a scalable and efficient reliable file distribution protocol. The aim is to achieve a system whereby a computer can choose to distribute a file or set of files to other clients on a network, for example for synchronous purposes. The system should keep errors minimal and as the system grows, for example if the number of nodes on the network increase, then the performance of the protocol should only be minimally affected, if at all.

2 Design

2.1 Decisions

There are some important factors that need to be considered before making concrete design decisions for the protocol. We need to consider :

- Time efficiency - When a file is to be distributed, it needs to not only go from one end point to another quickly - we also need to consider the time efficiency of the system when there are multiple nodes on the network to distribute the file to.
- Space efficiency - The system should do its best to minimise the size and number of packets it uses during packet transmission. We also need to consider how much of the file is kept in memory and how much should be read from/written to files during the transmission process.
- Correctness - Errors in transmission need to be kept to a minimum, but if there is potential for them anywhere, then they need to be spotted and corrected. Any system of error correction needs to be done efficiently and with as little interruption as possible if any in the rest of the transmission protest.

2.1.1 Centralised vs Decentralised

A centralised structure could lead to a few different solutions, depending on how far centralised a solution should be. For example, BitTorrent is a partially peer-to-peer network. However, it incorporates the use of tracker machines, which keeps a track of nodes which contain the desired file. The tracker has minimal interaction with the setup. This is useful, as it means that the only traffic going between the tracker servers and the client machines are small bits of address data as opposed to whole files. It also means that locating peers is very easy, since all clients have the common interface location of the server itself, meaning that joining peer-to-peer networks and locating other peers on the network doesn't rely on the client searching the network, but simply asking the server for information. This does however shift the responsibility of network speed onto the client machines.

On the other end of the spectrum, some file servers have all of the files uploaded and downloaded to central locations. This means that although the files are very easy to find from any node that if the file servers come under a lot of traffic then the network becomes congested and the system inefficient, particularly with multiple uploads and downloads.

A fully decentralised model would mean that there is no central weak point for where traffic will congest. Peer-to-peer performance scales logarithmically as more machines join a network, since the amount of data that has to be transferred is constantly reduced. For example, if a peer-to-peer network has 2 machines on it, one receiver and one sender, the sender has to send the whole file. However, if there are 2 senders, then each can send half of the file, instantly improving performance. These performance increases continue as more machines are added to a network, however the size of the increase decreases as more machines join.

Conclusively, we can see that peer-to-peer is an efficient mechanism, as the work load can be distributed to a number of different machines. However, a central tracker server would be useful in a small peer to peer network, as if the network is small the traffic will not be too much of an issue, and it would save the individual nodes the task of finding other nodes on the network to get files from.

2.1.2 Unicast vs Multicast

A unicast mechanism involves either a TCP or UDP connection from one machine to another. Since it has the potential for using TCP, it is useful in guaranteeing that one packet reaches another. A system can carry out concurrent unicast connections using threading, however this can be difficult to setup, and without care can lead to difficulties in the system. For example, there could be problems with read-write cases, and any deadlocks in communications. As more threads increase, their performance will decrease too, meaning a lower performance in the file transmission also. Without threading, the system has to work by individually handling each connection, which will not work as a system scales.

A multicast mechanism can only communicate over a UDP connection. However, multicasting means that a stream of data can be sent to multiple nodes simultaneously without the need for setting up any threading. Therefore, it would be easy, with the correct measures for handling any UDP errors, to send data to multiple nodes simultaneously. This is a very scalable system, as adding more machines onto the file transmission network has no effect on the performance of the server machine, other than the overhead that comes from potentially needing to send out more missed packets as more nodes are on the network. Efficient file protocols such as FCast use multicast for these reasons. There are a variety of ways in which it can be done - such as broadcasting sequential file data in a round robin format until all clients confirm that they have received the data, but we can also plan it more carefully to be more efficient.

Therefore, if the UDP streams are handled correctly, then multicasting is a far more efficient and scalable system than multicasting, and would mean gaining the benefits of communications with multiple nodes without the dangers that come with multithreading.

2.1.3 TCP vs UDP

Using TCP guarantees that data is transmitted correctly from one place to another, as the protocol deals with missed and broken packets at the transport layer by handling periodic acknowledgement messages. However, this combined with the complexity of the packet means that there could be some unnecessary overhead that we may be able to deal with at the application layer - for example, acknowledgements over UDP.

By using UDP, we cannot always guarantee that a packet will arrive correctly or at all. However, we can add additional information to our packets, such as sequence number, so that a target node knows which packets it has missed in a sequence of packets, and we can implement our own acknowledgement requests. This means that you have the safety of TCP if the system is designed well, but without the overhead that comes with the complicated TCP packets.

Since the UDP datagram is a small size, it will have less strain on a network compared to a TCP connection, which could be useful in our protocol for large data transfers to lots of nodes on a network, such as when transmitting file data over multicast. However, we may not always be able to guarantee that a node receives certain critical messages, such as initial handshakes, without acknowledgement requests, and by that stage, implementing this carefully could be a case of basically reimplementing TCP in our application over UDP. Since these only happen occasionally and we want to guarantee that these go through correctly, it may be better to do these messages over TCP.

2.1.4 Extra Considerations

Compression is one way in which the system could be optimised, as it would mean that for large uncompressed files that fewer data packets would have to be transmitted. The SCP program has the option to compress files and it does indeed increase performace in some cases, such as the one I previously mentioned. However, it is redundant for already compressed files, and there will be temporal overhead in compressing the files to the point where it will save little time if any in compressing the file in the first place.

Security has a few different properties in data communications. In a system of network communications, it is important to consider :

- Data being tampered with while in transmission and trying to minimise this. Usually, a pre-agreed hash function can produce a checksum which can be transmitted to the client and used from there to figure out if any data was tampered with or corrupted during transmission.
- The authenticity of the data - i.e we need to make sure that a client cannot be fooled into thinking that data they have received has come from a certain user, when really it is from someone in the middle. This can be done with a certificate scheme, which on a local network for file transmission can be hard-wired into the local network.
- We want to eliminate the chance of data being intercepted and read whilst in transmission. Schemes that exist to help with this can include public-private key cryptography and this would suit well on a local file transmission network, where public keys can be hard wired onto the network, meaning they can be assumed to be authentic, as opposed to having to find a means of transmitting the public keys during file transmission.

2.2 Decision Conclusions

After considering the above, I have designed my system using features below.

2.2.1 General Procedure

- Server tries to connect to all of the hosts in the config file and forms a list of control connections. The server then sends the initial file packet over the establised TCP connections individually, containing filename, future uuid to use to identify session, and checksum of the whole file.
- Client reads this control request after accepting the connection from the server. It then sets up a multicast listener on the port and address in the config file and sets up the file descriptor to write to during transmission. It returns a response packet to the server, containing the uuid that the server sent.
- While all clients listen, the server broadcasts on the multicast address and port in the config file the chunks a number of chunks, specified by the sequence limit in the config file. Each chunk has a specified chunk id so that clients can keep track of which packets they have and calculate which ones they have missed.
- At the end of broadcasting the sequence, the server asks each client individually over their control socket if they have received all of the chunks from the sequence and the client returns a list of all of the chunks that they are missing. Once the server has done this for all clients, it broadcasts all of the packets in the list and repeats the process of asking the clients and getting another list of missed packets until all clients verify that they have the whole sequence. While the client receives chunks, they write them to file at an offset from the start of the file calculated from the size of a chunk and a sequence so that even if chunks are missing, chunks can still be written. Both client and server use the *lseek()* operation to reposition the OS's file pointer.
- After this has been repeated for all sequences of the file, the server sends an end of transmission message over the TCP control socket to each client. Each client calculates a checksum from the received file and compares it to the received checksum at the start and determines if the transmission was successful or not on that basis. It sends back a message to the server stating whether or not it was successful in transmitting the file based on the checksum results.

2.2.2 Use of Multicast

The system will use multicast for data transmissions. Having read the paper on the implementation on "Fcast Multicast File Distribution" by Jim Gemmell, Eve Schooler and Jim Gray, I thought that a multicast system not only reduces network traffic by not transmitting repeated data packets to different sources, but also increases the speed by not having to go individually one by one as would be needed in a unicast scheme, or implementing a threading system.

Therefore, multicast UDP will be used for transmission of file data packets (i.e the data being read from the file to be transmitted).

2.2.3 Multiple Connections for Different Messages

I found that whilst experimenting with UDP that I would prefer to have a TCP connection control any control messages between nodes, for example, the initial message between a server and a client to let the client know of the initial details of a file such as the name and checksum.

I found that this is achievable using the POSIX select() function. It takes three lists, the first being connections to read from, the second being a list of connections to write to, and the third being a list of connections to check for for errors. In return, it provides three lists, one of connections that can be read from, one of connections that can be written to, and one of connections for transmitting errors. This is useful, as in Python it can be given a timeout to listen for and block connections, meaning that the UDP file data streams and the TCP control messages can all be queued by a client in list of ready to read connections by select. This eliminates the need for multithreading for the sake of parallel connections.

2.2.4 Accounting for missing Packets in UDP Multicast of File Data Chunks

In file data transmission, the file will be split up into sequences, which will be a sequence of chunks to be sent before the server asks clients which chunks of the file they are missing. The client will know which ones they are missing, since each file data packet will have a chunk id attached to it and the server will also say what the id of the last chunk in the sequence was, as although the sequence will have a fixed maximum level, we cannot guarantee that the file will full the last sequence with chunks.

2.2.5 Client Server Relationships

I will be using separate programs for the client and server. In the multicast system, every node could have a server to distribute files and also a client to listen for file distributions. Likewise, the server could also be a central structure that distributes files when they get added or updated to the network.

2.2.6 Packet Formats

I spent a long time trying to figure out how my packets should be formatted. Originally, I was thinking of sending a stream of bytes with my own key value mapping scheme. However, I found that constructing this kind of packet was difficult to write, and became very string based on what I picked for the keys. This also meant that parsing would be difficult as I would have to write my own parser to break down the packets into the key values, and any sub key values to differentiate between header values and packet values.

Ideally, the packets should be simple to construct to and from byte form, and easy to parse into a data structure. From this, I found the Python struct package. This includes the functions pack and unpack - the first function takes a format for how values are written into bytes, for example, 3 integer values, and then the values to convert into bytes with the given format. The unpack function does the reverse - it takes the format and then the sequence of packed bytes and returns the original values in the form of an easily addressable tuple. This meant that I would not have to do too much manual parsing and would only need to define the format of the packets so that both client and server know how all packets should be packed and unpacked. Since these are fixed size, I will use these struct formats for fixed size data, such as headers, and then for variable sized data, such as file names, I will use regular byte strings put on the end of the packet.

2.2.7 Protocol Codes

I will define my own protocol codes, such as message numbers, in a few different classes that will be common to both client and server. These will need to include :

- A number code which represents the type of a packet message, such as init packet or file data packet.
- A series of keys of where certain data lies in data packets - for example, an index of where in an init file packet the checksum of the incoming file is.
- The data formats of the structs of the packets so that they can be constructed and parsed by both client and server.
- A set of internal codes, more for debugging, that determine when a message comes in whether or not it is a valid message or not. For example, check that it is a message for the same protocol, and that the packet is readable by the parser.

2.2.8 Security Measures

The system will provide a system where the whole file has a checksum attached to it which is checked at the end by the clients. The server can send the checksum in the initial packet and then the clients can individually calculate the checksum when the file has finished transmitting and determine from the comparision between these two values whether or not the file has been transmitted successfully or not. I will use a CRC-32 scheme as this is a very quick way of creating a checksum through cycling over the entire file. This whole file checksum is a minimum requirement.

We could also implement individual packet checksums, however the transport layer should deal with making sure that incorrect packets are dropped, so I do not consider this a high priority.

I will also include a UUID in all the packets associated with a certain file so that in the future if we need to consider multiple file sessions then different packets will not clash for different files.

If I have time, I would be interested to experiment with encrypting the packets before sending them to test for how the encryption affects the time performance on a local model with public-private key cryptography, particularly since most file transfer schemes now are built on other security layers, such as SSH in SFTP and SSL in FTPS.

2.2.9 Potential Flaws

Since control messages have to be done on an individual basis, the clients have to wait on each other to finish transmitting control messages before their turn. The server times out and drops the connection with a client if they take too long to respond to a control message so that the whole process does not stop.

3 Implementation

All components have been written in Python 3. To run the program, the program *python3* must be installed. To run either the client or server on a host machine, the whole source code directory needs to be in the host machine. This can be done with *scp* or *sftp*.

3.1 Running

To start up the system, a set of client nodes must be started up before the server so that they can listen for server connections. To start up a client node, run the command :

```
$ python3 ClientTest.py
```

To start up a server, it needs at least 1 argument - the first is a filename to transmit. The second argument is optional, and allows the server to restrict how many connections it is listening to. This was mostly used for easier testing. If the second argument is not included, the server will by default try to connect to all of the hosts specified in the config file, but will be able to ignore any of those hosts if they are not running the client. An example command for sending *example.txt* would be

```
$ python3 ServerTest.py example.txt
```

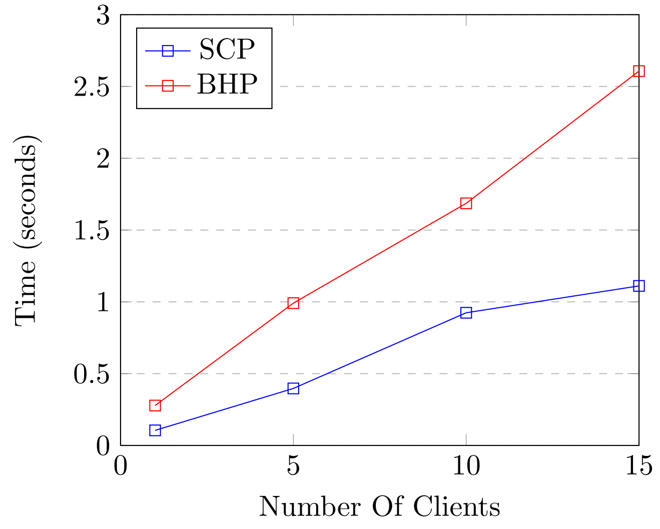
and an example for distributing it to the first 5 clients on the hostnames list, whether they are connected or not, is

```
$ python3 ServerTest.py example.txt 5
```

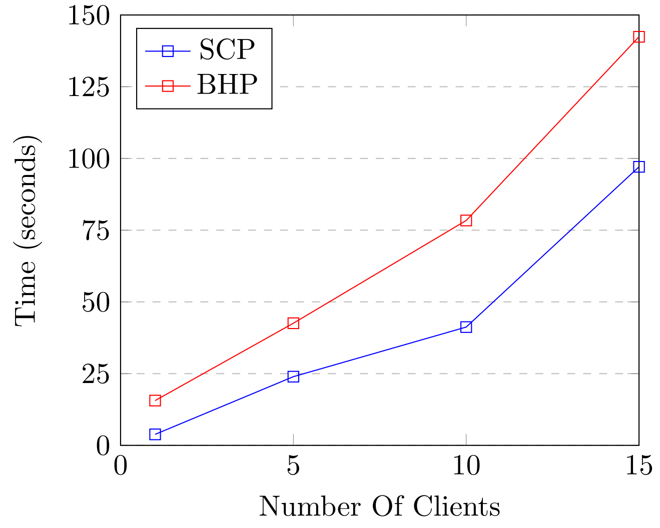
4 Results

The application has been tested and it does work. It can distribute files from one server to one client or from one server to many clients successfully. To compare it to a normal benchmark, I will compare how my protocol performs against SCP for different tasks.

Comparing Medium File Transfer (movie.mjpeg) to varying numbers of clients between SCP and My Protocol



Comparing Large File Transfer (Fast and Furious) to varying numbers of clients between SCP and My Protocol



5 Conclusion

Conceptually, my multicast protocol is good, however it needs improvements. It could be improved as a lot of time is spent individually talking to clients, so threading could repair this, and there is a large overhead with clients calculating checksums at the end of the process instead of incrementally throughout the transfer.