Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU33031 Computer Networks
Assignment 1: File Retrieval Protocol

Liam Junkermann, 19300141

November 1, 2022

Contents

Abstract

This report will discuss the implementation of a file retrieval protocol. Starting with a brief overview of a protocol which was used as inspiration, the components of the final implementation, an example of a network topology employing this implementation of the protocol, a brief discussion of potential improvements, and finally a summary and reflection of the assignment.

1 Introduction

For this assignment, a network containing one ingress server connecting a number of clients and workers, working together to request and provide files, was described. Files are to be requested from a `Client` via the `Ingress` node, the `Ingress` node then forwards that request to a worker, the worker then sends the file in response which is also forwarded through the `Ingress` node.
In this report I will discuss the inspiration for my solution followed by an explanation of overall design of the network, each element of the solution, and a description of the data packets and content transferred across the network.

2 Theory

At its core, this file retrieval network is a load balancer for file retrieval workers using multiplexed transport over UDP (User Datagram Protocol). QUIC (**Q**uick **U**DP **I**nternet **C**onnections) Protocol[1] is a protocol which implements this concept of multiplexed transport over UDP. A cursory understanding of this protocol provides the basis for a file retrieval protocol which handles multiple sized files as well as basic errors such as packet loss.

2.1 QUIC Protocol

QUIC was designed to improve page load times, while maintaining secure transport, as an alternative to TCP (Transport Control Protocol) + TLS (Transport Layer Security) + HTTP/2 (Hypertext Transport Protocol/2.0). QUIC was designed to be built on top of UDP as UDP does not suffer from the restrictions imposed by TCP protocols requiring system kernel operation, for example. Due to being built on top of UDP secure connections are maintained and other features are available such as:

- Reduced connection establishment time - 0 round trips in the common case
- Improved congestion control feedback
- Multiplexing without head of line blocking
- Connection migration
- Transport extensibility
- Optional unreliable delivery [1]

QUIC operates by establishing a single roundtrip handshake with a client, providing the client with the details to make future requests. From then on QUIC generally needs zero-roundtrips before sending future payloads. QUIC is able to maximise throughput by leveraging UDP's unreliability. Where TCP requires data streams to complete, UDP is nearly expected to lost packets. As a result QUIC is able to handle multiple requests without being blocked by the TCP head-of-line blocking. [2]

3 Implementation

3.1 Overview

This protocol uses a similar approach to QUIC to handle file responses and requests with a UDP protocol which utilises basic handshake mechanisms to achieve a basic protocol. This has been implemented using three components, configured in a virtual network. These components are an `Ingress`, a `Worker`, and a `Client`. A network would contain one `Ingress` node, one to many `Worker` nodes, and zero to many `Client` nodes. On startup `Worker` and `Client` nodes register with the `Ingress` which then enables the `Clients` and `Workers` to interact more efficiently.

[1] https://www.chromium.org/quic/

3.1.1 Packet Overview

The base packets have a two byte header where byte 0 is the packet type, and byte one is the source index. Byte one is not always used. The packet types, with their respective value are listed below

```
// Packet Types
static final byte FILEREQ = 0;
static final byte FWDFILEREQ = 1;
static final byte FILERES = 2;
static final byte ERRPKT = 3;
static final byte REGCLIENT = 4;
static final byte REGWORKER = 5;
static final byte REGACK = 6;
static final byte FWDFILERES = 7;
static final byte FILERESACK = 8;
```

Listing 1: Code snippet from `Node` with the encoded packet types

All packets transmitted through this network follow a packet structure detailed in Figure 1.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------------------------------------------------------+
|                 |                |                              |
|   PACKET TYPE   |  SOURCE IDX    |                              |
|                 |                |                              |
+--------------------------------+                                |
|                                                                 |
|                              PAYLOAD                            |
|                                                                 |
+-----------------------------------------------------------------+
```

Figure 1: Standard Packet

There is additional information encoded into the file response packet (`FILERES` and `FWDFILERES`) payload. In order to support files requiring multiple packets to send some extra information must be included with the file byte array content, therefore each payload has a 3 byte "header" of sorts. This "header" includes the sequenceNumber shifted right 8 bits, the sequence number (twice for error checking), as well as the End of File Flag which indicates to the client to stop expecting packets. This results in a packet describe below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----------------------------------------------------------------+
|                 |                |                |             |
|   PACKET TYPE   |  SOURCE IDX    |sequenceNo >> 8|  sequenceNo |
|                 |                |                |             |
+-----------------------------------------------------------------+
|                 |                                               |
|    EOF FLAG     |                  PAYLOAD                      |
|                 |                                               |
+-----------------------------------------------------------------+
```

Figure 2: File Response Packet

3.1.2 Operation overview

Figure 3 and Figure 4 show the processes to register and introduce new nodes to the network, and request and transmit requested files across the network respectively.

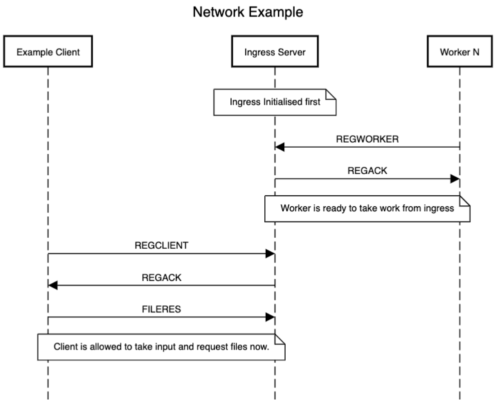

Figure 3: Registration of `Client` and `Worker` nodes to the `Ingress` node.

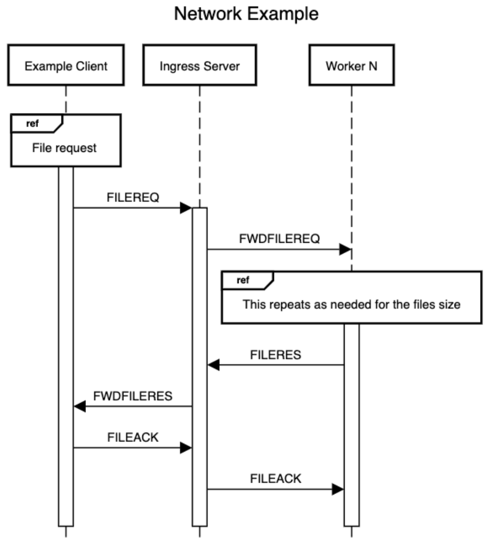

Figure 4: The sequence diagram detailing the process of requesting and transmitting files.

The implementation of this network allows for efficient communication between workers and clients through the `Ingress` node. The use of a map in the `Ingress` allows for a more efficient packet header as only 1 byte is needed to keep track of workers and clients. As all the traffic goes through the `Ingress` node, it can act as a router without massive impact on performance. The single byte packet header does impose a limit of only being able to serve up to 15 active clients or workers. If a larger network was needed this could be converted to a two byte header for source index to allow up to 255 `Client` nodes and `Worker` nodes. Given the size of this implementation 15 workers were more than necessary.

Figure 5: Client Interactions and logs from headless nodes

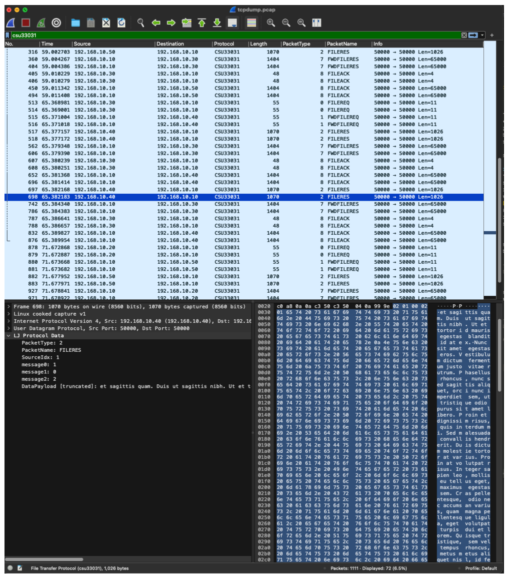

Figure 6: The network capture of file transfers between the `Worker`, `Ingress`, and `Client` nodes

Figure 5 shows an example of the basic operation of the entire network. The left pane shows the logs of the headless nodes (the `Ingress` node, and two `Worker` nodes). The two panels on the right show the input and output of example Client nodes. Figure 6 shows the block of packets which are doing the actual file transfer. You can see the forwarding of File responses and acknowledgments, as well as a peek into the final packet of the example lorem.txt file.

3.2 Network Components

3.2.1 Node

In practice each of these network nodes implement a Node class which handles most of the boilerplate Datagram behaviour and sets constants for header length, values, and positions, as well as providing some worker functions to generate packets and packet data appropriately. This Node also handles the multithreading needed to successfully multiplex within the Ingress.

3.2.2 Ingress

The Ingress works as a router or proxy. All requests from Client nodes pass through the Ingress node and are routed to the relevant Worker nodes, then again from Worker to Client. The Ingress node must always be the first node to activate as all the other nodes must be registered with the Ingress node. Registration is done in order to make the packet header a bit slimmer, as previously discussed. There are generally five packet types supported by the Ingress:

Node Registration There are two kinds of registration packets: Worker and Client registration packets, constants REGWORKER and REGCLIENT respectively. The registration packets register the Worker or Client in an ArrayList, the index of which will be used to reference the relevant node in future network events. The use of an ArrayList allows nodes which may have disconnected, or restarted, to be re-registered without growing an array.

File Request The FILEREQ is a request made from any Client node which is then processed and forwarded to a Worker in a FWDFILEREQ packet which contains information such as the source index to allow the response data to be directed appropriately. The worker is selected using a simple round-robin approach.

File Response The FILERES is the response to a File Request from any Worker node forwarded to the relevant Client, based on the ArrayList index in the header, in a FWDFILERES packet which includes the workers source index in the header.

File Response Acknowledgment The FILERESACK packet is a key part of the file requesting/response flow. For large files requiring multiple packets to transfer this acknowledgment packet allows the worker handling a given file transfer to progress to transmitting the next packet. The Ingress node handles this packet by replacing the source index value with the index of the sending Client, before forwarding the rest of the packet, untouched, to the relevant Worker.

Worker Error Finally, the ERRPKT is a packet sent from a Worker node to be forwarded to the Client node. As will be discussed in the Client section, the Client essentially blocks any input until a request has been completed. If an error occurs in the Worker this packet is sent, via the Ingress, to the blocked Client.

The Ingress node is a headless node, meaning it does not take user input. As such, it is one of the nodes, when deployed using Docker Compose[1], which is automatically started when the cluster starts.

3.2.3 Worker

The Worker node handles retrieving files from the filesystem based on requests from Clients via the Ingress. The main work done by the Worker node is through the sendFile function. This function fetches the requested file from the filesystem, converting it to a byte array, then iterating through that byte array in 1021 byte blocks until the end of file, transmitting each block with some extra wrapping to let the Client know which "sequence" number has been sent, and if the Worker has reached the end of the byte array. The final FILERES payload size becomes 1024 bytes with this extra information added. This packet is then wrapped to include the information

[1] Docker Compose: https://docs.docker.com/compose/

```
(base) ➜  fileRetrievalProtocol git:(master) ✗ docker-compose -f file_retrieval.yml up -d
[+] Running 7/7
 ⠿ Network fileretrievalprotocol_file_retrieval  Created                    0.1s
 ⠿ Container tcpdump                             Started                    0.6s
 ⠿ Container worker1                             Started                    1.1s
 ⠿ Container client1                             Started                    1.1s
 ⠿ Container ingress                             Started                    1.2s
 ⠿ Container worker2                             Started                    1.7s
 ⠿ Container client2                             Started                    1.8s
(base) ➜  fileRetrievalProtocol git:(master) ✗ docker-compose -f file_retrieval.yml logs ingress
ingress  | [Node] Instantiated node
ingress  | [Ingress] Instantiated new node
ingress  | Starting ingress program...
ingress  | Registering new worker /192.168.10.40:50000
ingress  | Sent RegAck Packet
ingress  | Registering new worker /192.168.10.50:50000
ingress  | Sent RegAck Packet
(base) ➜  fileRetrievalProtocol git:(master) ✗
```

Figure 7: A snippet of the Docker Compose Logs showing the `Ingress` node service starting automatically

the ingress needs to forward the data appropriately. The `Worker` then waits for the acknowledgment packet to be returned before continuing onto the next packet of data (if applicable).
The packets which the `Worker` node's `onReceipt` function handles are:

Forwarded File Request The `FWDFILEREQUEST` comes from the `Ingress`, as noted above, and triggers the `sendFile` resulting in the behaviour described above.

Registration Acknowledgment The `REGACK` also comes from the `Ingress` and lets the worker know it is registered.

Just as the `Ingress` node is a headless node, so is the `Worker` node. As a result both workers get launched automatically when the cluster starts.

```
(base) ➜  fileRetrievalProtocol git:(master) ✗ docker-compose -f file_retrieval.yml logs worker1 worker2
worker1  | [Node] Instantiated node
worker1  | Starting worker worker1 program...
worker1  | Sent reg packet with data
worker1  | File Retrieval Protocol starting...
worker1  | Successfully registered worker with ingress server
worker1  | ingress:50000
worker2  | [Node] Instantiated node
worker2  | Starting worker worker2 program...
worker2  | Sent reg packet with data
worker2  | File Retrieval Protocol starting...
worker2  | Successfully registered worker with ingress server
worker2  | ingress:50000
```

Figure 8: A snippet of the Docker Compose Logs showing the `Worker` node services starting automatically

3.2.4 Client

The `Client` node handles taking user input and printing out the result of file requests. Much like the `Worker` the heavy lifting of the function is done by the `handleFileRes` function. First, a user enters a filename input. This action triggers a file request to be sent to the `Ingress` node and blocks further user input. The `handleFileRes` function handles the response packets sent from the `Worker` nodes through the `Ingress` node, writing the byte content to a buffer until the finished flag has been sent. Once the end of file packet has been sent, the `handleFileRes` function then prints out the `fileByteArray` in string format, clears the buffer, and unblocks the user input to allow a new file to be requested. While handling the file response packets, the

`handleFileRes` function validates the packets have been received in the right order, and responds with the appropriate acknowledgment packet.
The remaining packets handled by the `Client` node are:

Error Packet The `ERRPKT` packet is sent from the `Ingress` node when an error has occurred within the `Ingress` or `Worker` node for a given transaction. This packet is sent to unblock the `Client` so they user may continue by either trying to repeat their request or suggesting them to restart the network.

Registration Acknowledgment The `REGACK` packet is sent from the `Ingress` node to confirm that the `Client` has been registered and can request and accept files.

The `Client` node requires user interaction. The current Docker Compose environment creates the virtual machine and a user must start and interact with the Client once the cluster has started.

Figure 9: An example of a user using the `Client`, showing an error, a small single-packet file, and a larger multi-packet file transaction

4 Future Enhancements

4.1 Load Balancing

The `Ingress` node also acts as a load balancer for the workers. The load-balancing approach used has much room for improvement. Currently a simple round robin approach is used, iterating a counter with each new file request to select which `Worker` to use. This could be improved by being more dynamic with load balancing, including using the forwarded packet sizes, or even the content (has the `Worker` sent a packet with an EOF flag in the last `x` packets), to determine if a worker is getting requests for larger files. This load balancing approach also does not account for Nodes disconnecting after registration. Having a mechanism to track if a node is still online could be helpful in maintaining `Client/Worker` maps.

4.2 Node Registration

The `Client` and `Worker` nodes could be improved by building a more robust registration process. With the current implementation it is imperative that the `Ingress` node is enabled and active before new nodes may be introduced to the network. For the purposes of this project I introduced a `sleep` function call to the initialisation of the `Worker` nodes, as those are brought online by the docker compose script automatically. The `Client` nodes do not need this workaround as they are started manually once the headless nodes activate.

5 Summary

In this report discussed the requirements for the assignment, the real-world version of a solution from which I drew inspiration, and finally my implementation of a file retrieval protocol. Then, an exploration of the basic functionality and an example topology of the completed network, as well as some limitations and potential improvements to this implementation.

6 Reflection

This project is relatively large in scope, and having biweekly "check-ins" as blackboard assignments has helped break down the project and remind me to continue working on it at a more manageable pace. While these check-ins were helpful, it was not entirely clear what the objective of each one was and will be something I will clarify for the next project. In general I find it much easier to understand concepts discussed in lectures when I have the opportunity to work with them. Working on this project has given me a better understanding of the basic networks concepts discussed in class, and showed me how a design decision earlier on in the process can have knock-on effects when implementing more advanced features such as file requiring multiple packets to transmit completely. Without including the time needed to write this report this project took me roughly 24 hours to complete. I spent a bit more time earlier on exploring the QUIC protocol to get a better understanding of how to implement a similar solution. In retrospect it may have been more efficient to try and put something together to get a better understanding of the potential problems and iterate on that approach. I noticed I was able to understand the documentation associated with the QUIC protocol much more after completing a version of my solution, based on the learnings I had while building the solution.

References

[1] Google, and Internet Engineering Task Force. "QUIC, a Multiplexed Transport over UDP." www.chromium.org, Google, 2022, www.chromium.org/quic/.

[2] Jana Iyengar and Martin Thomson. QUIC: A UDP-Based Multiplexed and Secure Transport. RFC Editor, 2021, doi:10.17487/RFC9000.