Live Modifiable Server

by

Himanshu Mundhra

(16CS10057)

A thesis submitted in partial fulfilment for the
degree of Bachelor of Technology in
Computer Science and Engineering

Under the guidance of
Professor Sandip Chakraborty
Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Spring Semester, 2019-2020

DECLARATION

I, Himanshu Mundhra, certify that this thesis titled "**Live Modifiable Server**" and the work presented in it are my own. I also confirm that

- The work contained in this report has been done under the guidance of my supervisor while in candidature for a Bachelor degree at this University.
- The work has not been submitted to any other Institute for any degree or diploma.
- I have conformed to the norms and guidelines given in the Ethical Code of Conduct of the Institute.
- Whenever I have used materials (data, theoretical analysis, figures, and text) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references. Further, I have taken permission from the copyright owners of the sources, whenever necessary.

Date: June 11, 2020 (Himanshu Mundhra)
Place: Kharagpur (16CS10057)

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Department of Computer Science and Engineering

Kharagpur - 721302, India

CERTIFICATE

This is to certify that the project report entitled "**Live Modifiable Server**" submitted by **Himanshu Mundhra** (Roll No. 16CS10057) to Indian Institute of Technology Kharagpur towards partial fulfilment of requirements for the award of degree of Bachelor of Technology (Hons.) in Computer Science and Engineering is a record of bona fide work carried out by him under my supervision and guidance during Spring Semester, 2019-2020.

June 11, 2020
Kharagpur

Professor Sandip Chakraborty
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Abstract

Professor Sandip Chakraborty

Department of Computer Science and Engineering

Bachelor of Technology

Live Modifiable Server

by Himanshu Mundhra

We have build a robust system from scratch on C++ which envisions to add scalability and dynamism to existing Server-Client Connection Systems. We build over here, a concurrent server with a thread-per-connection concurrency-control mechanism. The Control Channel listens and accepts new connections, which are each then handled by a forked child called the Connection Channel. As the Connection Channel handles the connection, the Control Channel continues to accept connections and delegate them to forked Connection Channels.

Within the Connection Channel, a Data Channel is spawned using an execvp() and acts as an instance of data transfer between the Server and the Client. We develop an elaborate internal and external communication protocol which enables us to pause every connection (thereby terminating this instance), modify the executable of the Data Channel, and then re-spawn every instance of the Data Channel with the modified image - which resumes the connection from the previously paused state.

Acknowledgements

There are many people who I would like to thank as I pen down (or rather type out) my thesis. As my final semester started, I had plenty of plans both academic-research oriented as well as recreational. Things seemed all set for a grand end to a well rounded college life. But Alas! these unprecedented circumstances. I guess it are these life-threatening overtly challenging situations which give us some perspective and a realisation to truly acknowledge the valuable contributions which make our life easy.

I would like to thank my family for continuing to provide for me in these testing times and allow me to have the head space to actually be able to build this system from absolute scratch, and debug it right down to unaccounted bytes.

I would like to thank my friends for providing me with constant mental support allowing me to plough through my problems and find solutions on my own.

I would especially like to acknowledge the relief workers who worked continuously to revert the wreckage which the Super Cyclone Amphan caused to Kolkata, and my neighbourhood in particular. I would like to thank the Dean(UG) for his kind consideration to allow me some more time to complete my BTP-II in light of the disruption caused due to Amphan. Special thanks to my HOD - Prof. Dipanwita Roychoudhury, my FacAd - Prof. Sourangshu Bhattacharya and my panelists - Prof. Sudeshna Sarkar and Prof. Partha Bhowmick for understanding the gravity of my situation.

And last but not the least my guide and advisor Prof. Sandip Chakraborty for his unwaivering belief in my abilities, his valuable pointers to set me on course and in general being the kind and helpful person that he is.

I definitely could not reach the goals I had set at the start of the semester, but I am proud that I could build an end-to-end fully functional scalable system.

Contents

Chapter 1

Introduction

1.1 Motivation

The Live Modifiable Server is a regular data transfer server-client system which can cater to client requests of GET and PUT; getting data from the server side and putting data on the server side respectively.

But what makes it special ?

There are instances when we want to tweak and modify the way the server handles connections and deals with data transfers. There might be a need to block some connections, there might be a need to changes the access permissions to certain data stores, there might be a need to redirect data from existing channels to new channels.

But how to go about doing this online while connections exist and are being catered to ?

The easiest thing to do is simply disconnection from all the connections while the modification takes place in the server. Even though this ensures that there is no delay in incorporation of changes with respect to the data transfer timeline, there is a significant real-time delay due to this downtime at the server end. Moreover, handling the re-connection of the clients is another headache. The re-connection has to be initiated by the client, and there is no way to accurately predict the extent of the downtime. This would just mean that the client keeps polling on the listening port of the server at regular intervals until it establishes the connection. Depending on the implementation of the protocol, there might be a need to store the state of the transfer before the downtime began so as to resume the transfer from where it was halted previously.

This is where our system comes into the mix.

We develop an elaborate internal communication scheme between the Control Channel (open to the world to accept connections and signals) and the Connection Channel (specifically to cater to a client, surviving across server downtime) which allows the system to continue catering to the client from the previously paused state after resumption of services at the server side. We build a wrapper over our conventional sever, which stays alive throughout the code modification process; not only preserving the existing connections and their state but also readily accepting new connections and queuing them until server modification finishes. This ensures that the issue of client re-connection is mitigated since once connected, the server downtime has no real affect on the validity of the connection. More so, since the wrapper is forever alive, the ease of storing the state of the connection also increases.

1.2 Use Cases

1.2.1 Replication Channels

In huge companies which work on Terabytes of data regularly, data backup is a big issue. Consistent and continuous data backup is required to ensure no loss of data occurs, because there are instances where reproduction of data is not possible. With the advent of Cloud Storage and Distributed Systems, Replication channels have come into being which provide this facility. Multiple nodes are registered to which streams of data are sent to get backed up. In some setups, backup data is sent to all the nodes to increase the fault tolerance. However this has its own penalty on the network traffic and considerably slows things down. A variation to this approach would be to choose target nodes and be able to update them dynamically as and when required. Node maintenance or re-routing of data in the network can all be accounted for with this wrapper on top of the conventional channels. A small scale version of this functionality has been demonstrated in the upcoming chapters.

1.2.2 Version Control

In large scale software which are used by millions of users worldwide, updates and fixes keep on getting released. However the issue of rolling out these newer versions is a tacky one. As of now, these updates are rolled out in batches and incrementally reach the whole consumer base. We envision that usage of our system may speed up this process.

1.2.3 Virtual Network Funtions

These days networks consists of chains of Middleboxes and the execution of the Network functionalities occur over these chains. These Middleboxes may contain what we call Virtual Network Functions which carry out different network functionalities. There might be instances where we need to modify these VNFs dynamically.

For example; for the end to end processing of packets, it needs to pass through some network functions - say 4 of them. Now out of the 4, one is getting modified while the other 3 remain put. What to do in this scenario ? Typically the current idea is to pause the application, make changes and then resume the application.

At IIT Kharagpur, between our ISP - the .ernet network and the .iitkgp network, there are a set of middleboxes - Firewall, NAT, Login Service etc. through which packets pass. For the time being consider that these applications are all Network Functions; typically Network Functions are at a way more granular stage. Even inside a firewall there can be multiple network functions operating. So how to upgrade ?
The current way to upgrade any one of these middleboxes is to announce a network upgrade and suspend services for some time. The services resume once the upgrade is complete. This is something which CIC IIT Kharagpur does.

The main problem is the cross dependency amongst these middleboxes. If we want to modify one middlebox we actually have to stop the others; we cannot directly switch to a new one.

So the idea here was as follows -
When we want to upgrade some functionality, rather than stopping it, as and when the service is running we can partially suspend it and keep the other middleboxes running. If one function changes - I shall update that function and then resume it. Rather than pausing the entire end-to-end service, we can simply resume the service from the point where it was last executed by preserving the state.

The Advantages of such an architecture is that there is not fixed downtime - no need to impose restrictions and boundaries on the user. In case the user does not save their proceedings and continue to use the service as it goes into downtime, a crash can cause a lot of loss of data. Here before migrating to the modified version, a checkpoint of all connections from users can be maintained - which can be used to resume the connections from the previously paused state.

Chapter 2

The Different Actors

Description

There are various actors and entities involved in the functioning of the Live Modifiable Server System, each of which have their own indispensable roles to perform. We describe each of these actors in this Chapter.

2.1 Control Channel

The Control Channel acts as the backbone of the System. It is this process to which all the clients (BACKUP_Nodes or GET/PUT Clients) connect to.

Backup Nodes It initially accepts connections from all the Backup Nodes, registers them locally and sends out Confirmation INFO Packets to the Nodes. It continues to listen to Backup Nodes until a Timeout occurs or the Admin signals it an EOF with regards to incoming connections.

```
while (No System-Timeout && No Admin-EOF) {
    Accept_Connections(BACKUP_Node)
    Register(BACKUP_Node)
    Send_Confirmation(BACKUP_Node)
}
```

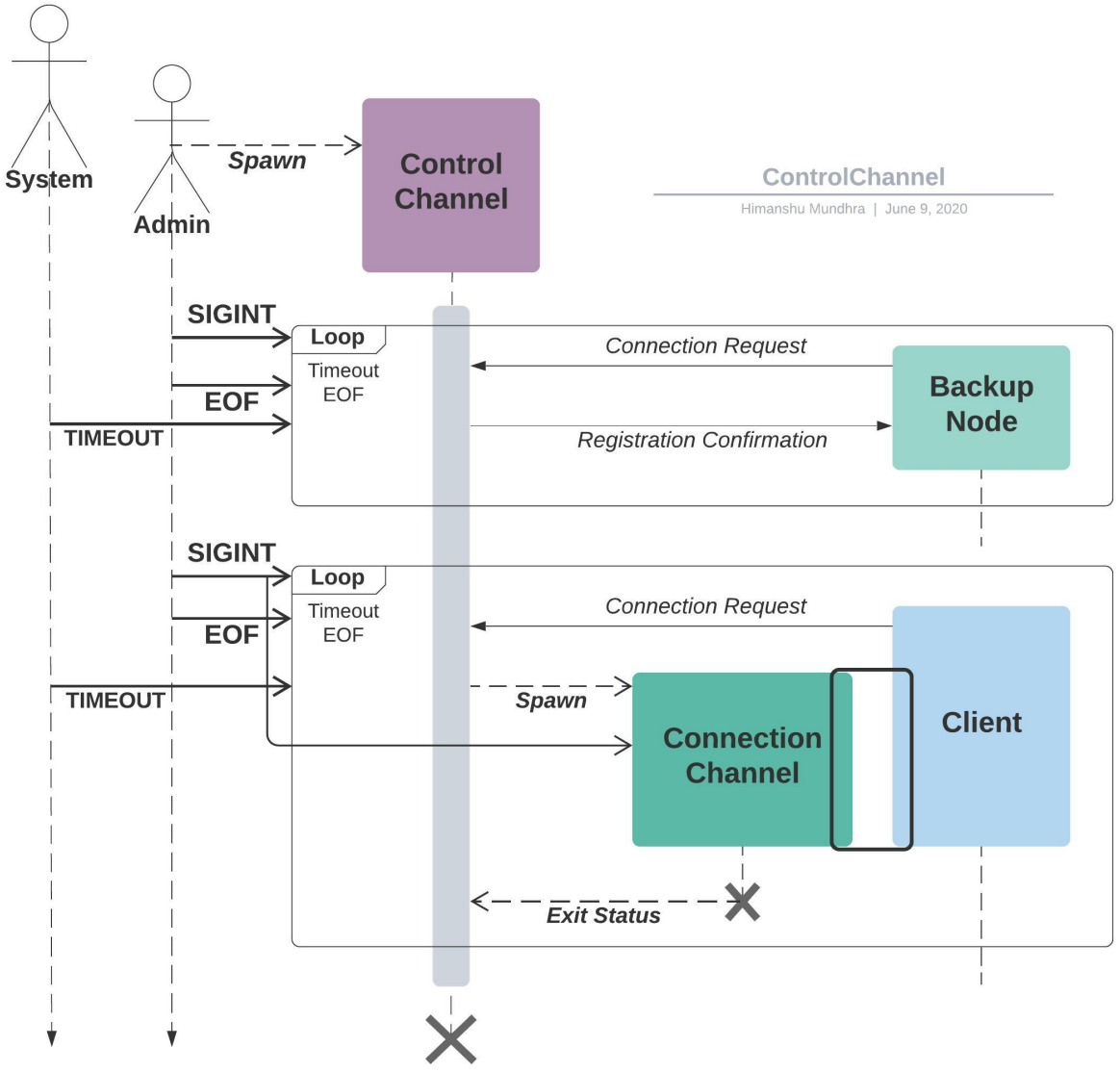

FIGURE 2.1: Sequence Diagram of Interactions at Control Channel

Data Clients It then listens and accepts connection from different GET and PUT Clients, forks a Connection Channel and delegates the responsibility of dealing with the Client to that particular Channel. This way the Control Channel is free to accept more connections. This is akin to the one-thread-per-connection concurrency control mechanism.

```
while (No System-Timeout && No Admin-EOF) {
    Accept_Connections(Data_Client)
    Fork(Connection_Channel)
    Delegate(Connection_Channel, Data_Client)
}
```

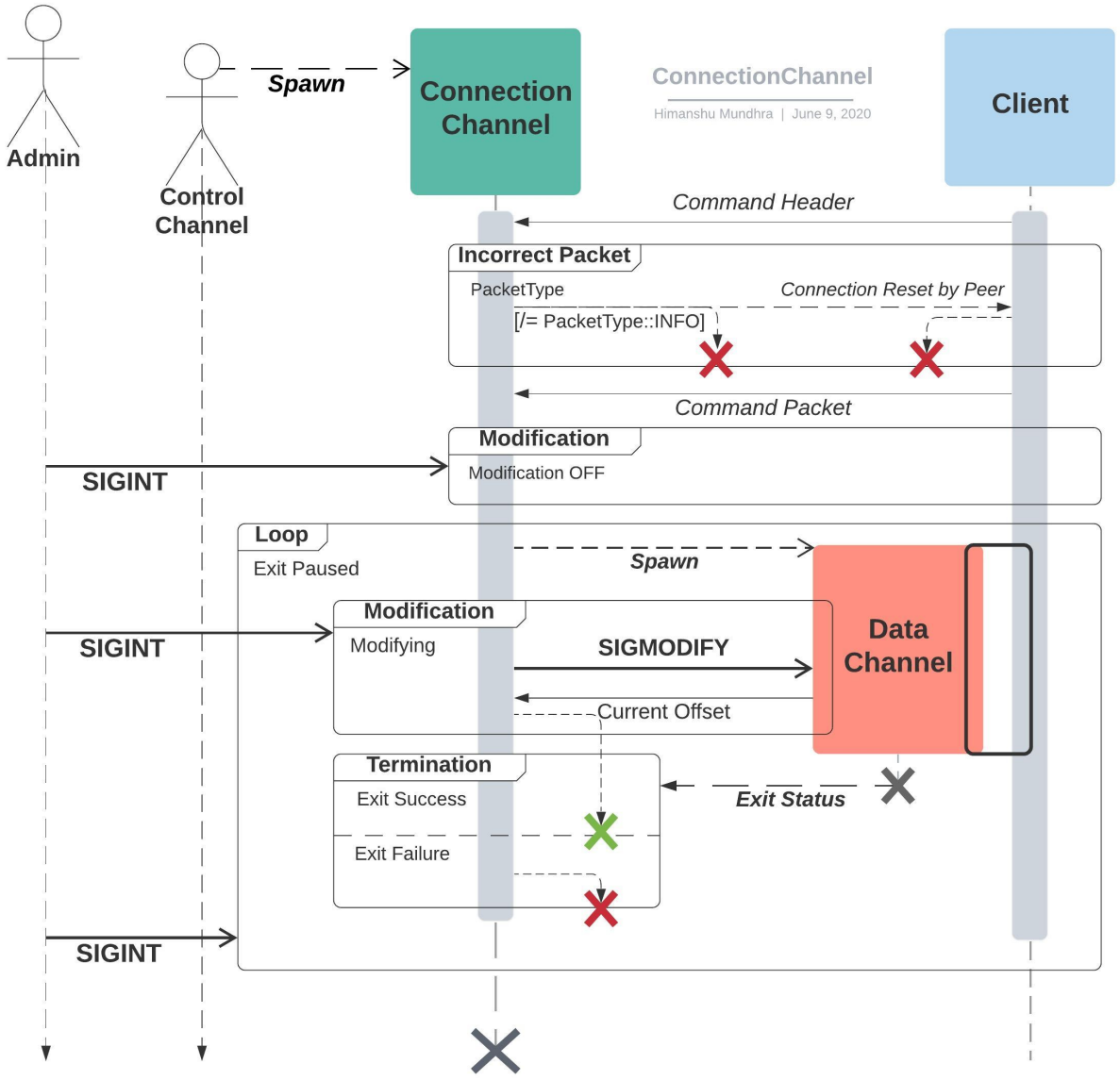

FIGURE 2.2: Sequence Diagram of Interactions at Connection Channel

2.2 Connection Channel

The Connection Channel acts as the henchman of the Control Channel, it is assigned a responsibility and it ensures that the task is brought to fruition. It takes care that the data transfer between the Server and the Client terminates - it might be a successful or failed data transfer.

Since the Connection Channel is forked by the Control Channel, it has the copy(-on-write()) of the complete address space of its parent. However there is no scope to modify the source code and come up with a new executable which can get reflected in the ongoing proceedings.

This is where the **Data Channel** comes into the picture. The Data Channel is a process spawned [*execvp()*] from the Connection Channel with it's own executable and address space.

It can be thought of as an **Instance of Data Transfer** between the Server and the Client - which goes on until the Transfer terminates or a Signal for a Modification is received.

It is this instance which is open for modification. The source code of the data channel can be modified and it's updated image can be pushed onto the ongoing proceedings - thereby providing our system with much needed dynamism.

```
recv(Command from Client)
do {
    if (ServerModifying) pauseFor(SIGINT)
    Spawn(Data_Channel)
    pauseFor(SIGINT, SIGCHLD)
    if (ServerModifying) {
        sendSignal(SIGMODIFY) -> Data_Channel
        recv(Updated_Offset) <- Data_Channel
    }
    ExitStatus = wait()
}
while (ExitStatus == PAUSED)
```

SIGINT (Ctrl+C or ^C) is an Interrupt which can be generated from the Terminal. The Admin uses the SIGINT Signal to alert the System of the start of the Server Modification Process. The SIGINT is to be sent to the whole process group (and not just the Control Channel) thereby alerting all the Connection Channels simultaneously.

```
SIGINT_SignalHandler() {
   ServerModifying = !ServerModifying
}
```

When the Connection Channel receives a SIGINT whilst a Data Channel is in progress, it sends a **SIGMODIFY** Signal to the Data Channel to terminate the transfer with a PAUSED Status. The Data Channel then returns an **offset** - which is the marker of percentage of data transfer completion. The Connection Channel then loops back and spawns a new Data Channel with this received offset.

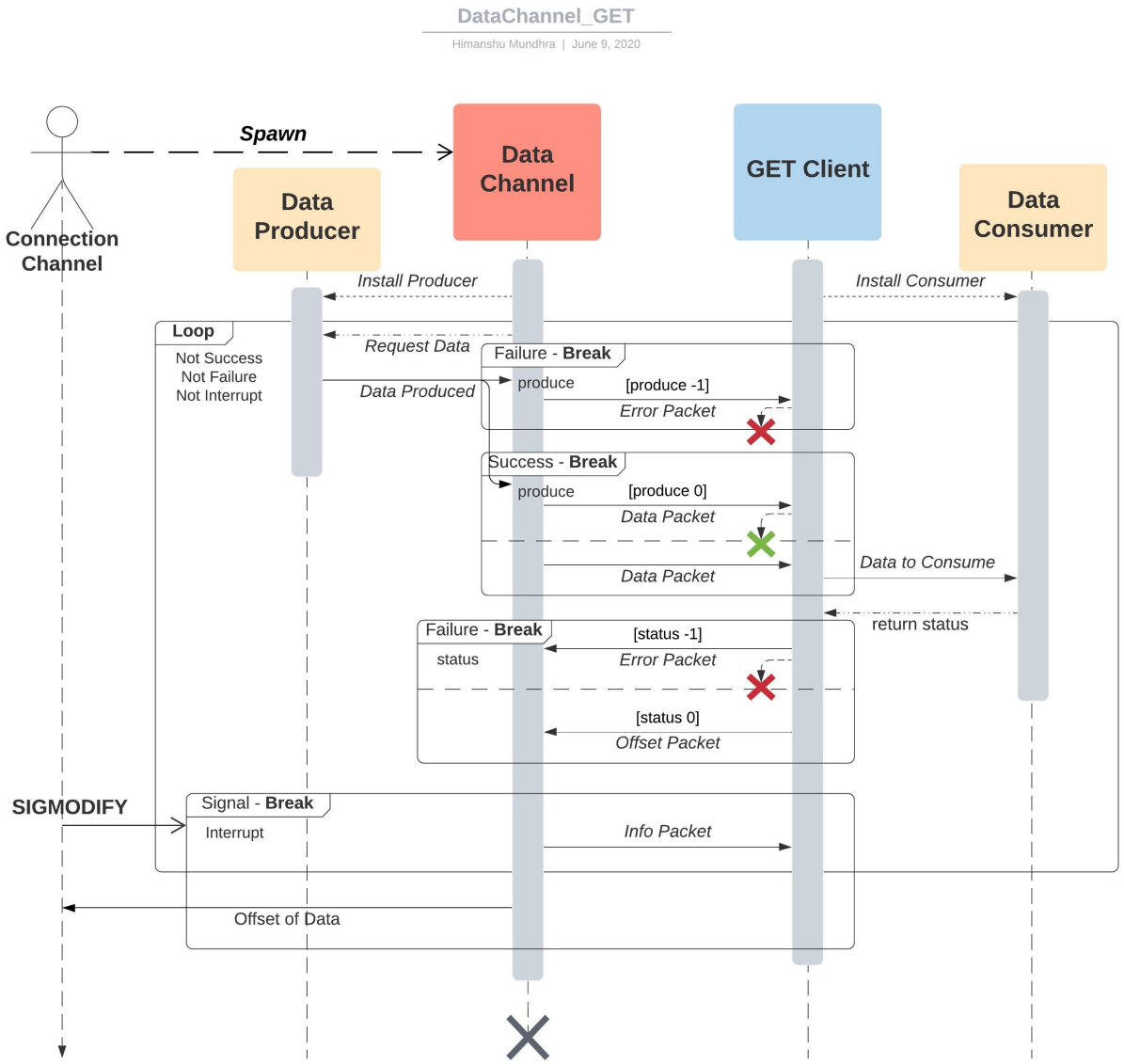

FIGURE 2.3: Sequence Diagram of Interaction between Data Channel and GET Clients

2.3 Data Channel and GET Clients

When connected to a **GET Client**, the Data Channel's task is to facilitate data transfer from the Server to the Client.

The Data Channel first installs the data producer on its side and starts requesting data from the offset provided by the connection channel (which is subsequently updated within the process itself).

It then sends the produced data to the GET Client, which then consumes the data as determined by it's needs as a Consumer. The Client then sends an Offset Packet which is basically a request to produce and send data from that offset. The offset at the Data Channel is thus updated.

GET Client It terminates successfully when data transfer is complete; and with an Error when the Server Side terminates the connection abruptly or the Consumer fails.

```
Consumer = Install()
while (No ServerError && No ConsumeError && Not Complete) {
    recv(Data) <= Data_Channel
    Consumption = Consumer.consume(data=Data)
    send(NewOffset) => Data_Channel
}
```

Data Channel This to-and-fro of data continues till the data transfer is complete. However, it may terminate prematurely due to Errors in Production or Sending, or on a SIGMODIFY.

```
Producer = Install()
while (No Error && Not Complete && No Interrupt) {
    Data = Producer.produce(offset=Offset)
    send(Data) => Client
    recv(Offset) <= Client
}
```

Interrupt at Data Channel The SIGMODIFY which the Data Channel receives from its parent - the Connection Channel associated with that Client basically sets a flag to terminate the ongoing data transfer with a PAUSED Status and return with the marked offset.

```
SIGMODIFY_SignalHandler() {
   Interrupt = 1
}
...
...
if (Interrupt) {
    send(Info) => Client
    send(Offset) => Connection_Channel
}
```

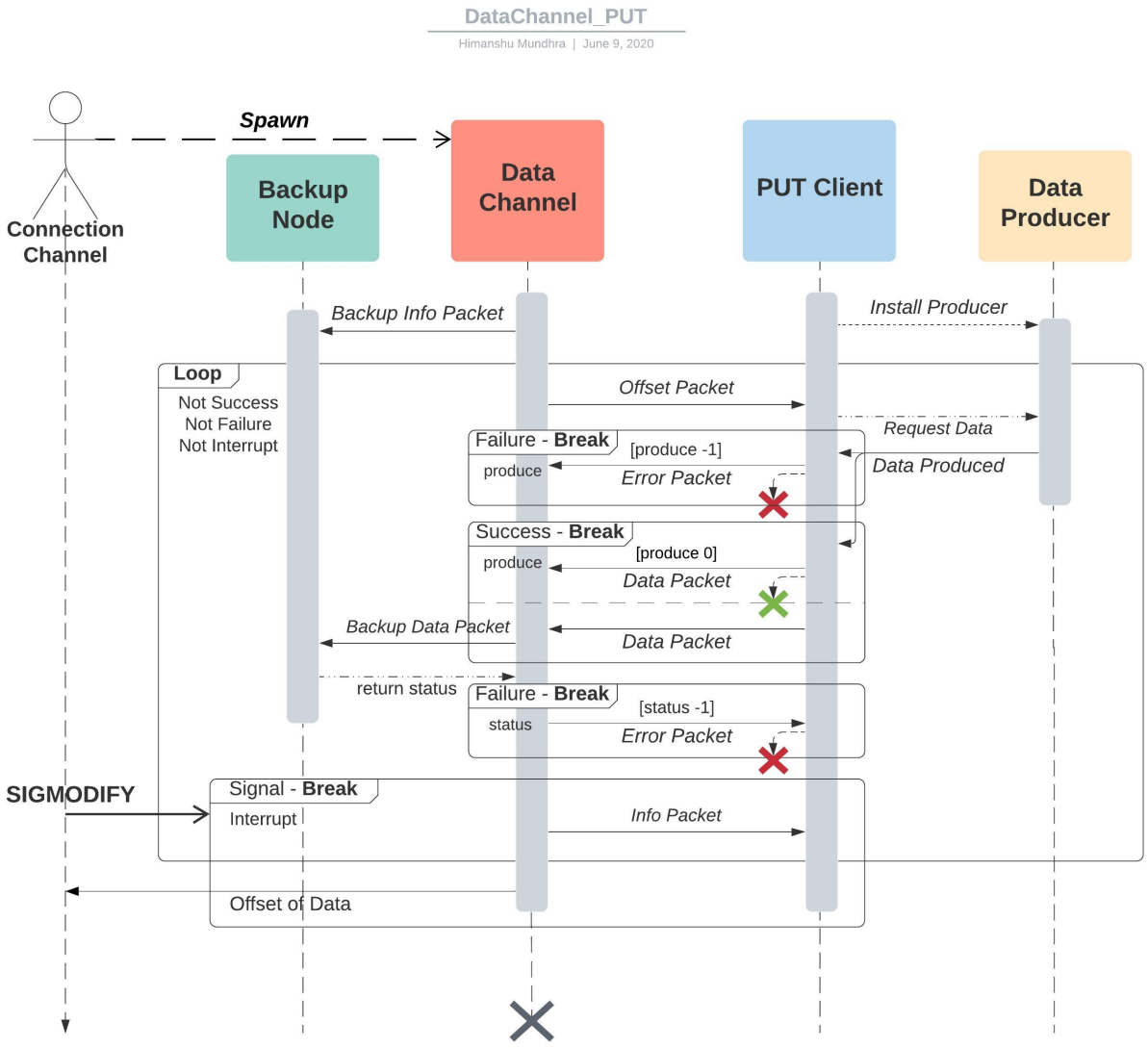

FIGURE 2.4: Sequence Diagram of Interaction between Data Channel and PUT Clients

2.4 Data Channel and PUT Clients

When connected to a **PUT Client**, the Data Channel's task is to facilitate data transfer from the Client to the Server to the BACKUP Node (if installed).

The Data Channel first sends a INFO Packet to the BACKUP Node in-use to inform it about the details of the data whose backup is going to be created.

It then initiates the transfer by requesting data from the client by sending an Offset Packet with the offset provided by the Connection Channel initially and updated subsequently.

The PUT Client, requests data from the producer by passing the required offset, and then forwards that data to the Server.

The Data Channel of the Server than forwards the Data Packet with some extra payload as a BackupData Packet to the BACKUP Node in-use. The offset is then updated as suitable.

Data Channel The termination of the Channel on Errors and Completion, and handling of the SIGMODIFY Interrupt remains the same.

```
send(BackupInfo) => BACKUP Node
while (No Error && Not Complete && No Interrupt) {
    send(Offset) => Client
    recv(Data) <= Client
    send(BackupData) => BACKUP Node
}
```

PUT Client It terminates successfully when data transfer is complete, and with an Error when Server Side terminates the connection abruptly or the Producer Fails.

```
Producer = Install()
while (No ServerError && No ProducerError && Not Complete) {
    recv(Offset) <= Data_Channel
    Data = Producer.produce(offset=Offset)
    send(Data) => Data_Channel
}
```

BACKUP Node It remains open to receive Backup Packets across Connection Channels. After receiving the Details in the BackupInfo Packet it registers the Connection Channel and appropriately stores the future BackupData Packets mapping them to the correct store using the Connection Channel PIDs.

```
do {
    recv(Backup Packet) <= Data Channel of any Connection Channel
    if (BackupInfo Received) Register Details of Expected BackupData
    if (BackupData Received) Store BackupData Appropriately
}
```

2.5 Producers and Consumers

These are entities which are installed externally at the Server and Client side during the data transfer part of our protocol.

Producers are installed at an host at the start of data transfer; they are invoked periodically to produce a chunk of data from the provided offset.

Consumers too are installed at the host in the beginning; they are invoked periodically to consume the chunk of data provided and return a marker of successful consumption.

An issue however which we might have overlooked here is the rewindibility of producers and consumers, and their mutual compatibility. **But what is Rewindibilty ??**

Rewindable Producers are a class of producers that have a notion of states and who can change their state on demand. In simpler words, a rewindable producer can *seek forward or backward* and start production of data from any point. This increases the complexity of the underlying implementation but on a less-reliable channel, the ability to produce data from the past in invaluable.

A **simple file** on a file system can be said to be a **Rewindable Producer** because we can use the *lseek()* function to seek to any point in the file, and then read data from there.

A **livestream of bytes** however, is a **Non-Rewindable Producer** since we can mostly not roll backward and produce past data again, and we most certainly cannot roll forward and produce data from the future.

Rewindable Consumers are the class of consumers which can request data from any offset - be it in the past or in the future. These consumers however need to have the provision of reverting back the changes in case they are going to re-consume some data, or in general be impervious to data re-consumption.

Compatability Not all producer-consumer pairs can be compatible with each other. For instance a Non-rewindable producer and a Rewindable Consumer are going to be incompatible. A Rewindable Consumer might request for an offset which is not corresponding to the current seek of the Producer, and being Non-Rewindable it cannot cater to that request.

Chapter 3

The Different Packets

Description

There are many different packets which do the rounds and are involved in the functioning of the Live Modifiable Server System, each of which have their own importance to the protocol followed by the system.

3.1 Error Packet

The Error Packet comes into use when an unexpected event abnormal to the correct control flow occurs - Socket Failed to Bind, Incorrect Filename, Data Production Failed, Incorrect Command Format and many others. As visible from the Sequence Diagrams above, after an Error Packet is sent or received, the Client crashes as Failed and the Server Side Channels terminate with a Failed Status.

Here is a description of what the different fields hold and mean -

Type - It holds the PacketType::ERROR which distinguishes it from other packets and intimates the receiver of which packet it is receiving.

Length - It holds an Integer Value which is the actual size of the Error Message which the sender has sent.

Error Message - Holds the string describing the error which occurred at the sender side.

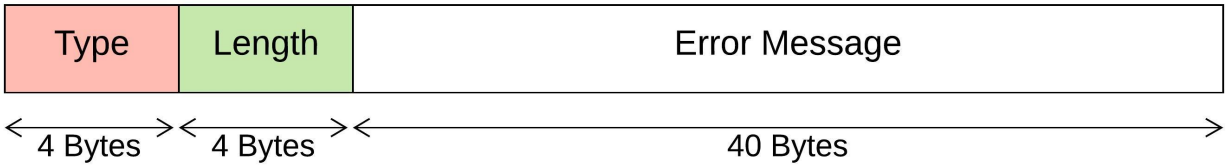

FIGURE 3.1: Error Packet

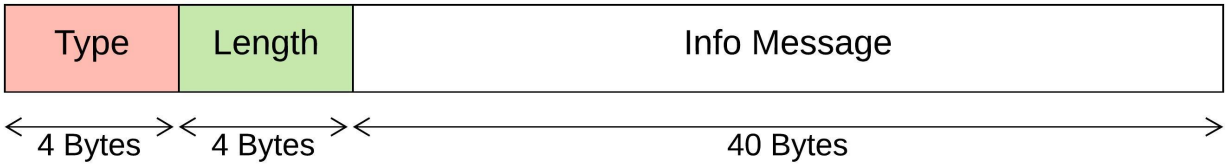

FIGURE 3.2: Info Packet

3.2 Info Packet

The Info Packet is a Multipurpose Packet which performs different tasks at different places in the System.

BACKUP Node connects to the Server, an Info Packet is sent to the Node to confirm its Registration.

Data Client connects to the Server, it sends an Info Packet which holds the Command to be performed in the Info Message field: **GET [Filename]** or **PUT [Filename]**.

Modification at Server side is taking place, the Data Channel sends an Info Packet to the Client to inform it about the delay in data transfer due to Code Modification taking place.

Here is a description of what the different fields hold and mean :

Type - It holds the PacketType::INFO which distinguishes it from other packets and intimates the receiver of which packet it is receiving.

Length - It holds an Integer Value which is the actual size of the Info Message which the sender has sent.

Info Message - Depends on the context of sending the Info Packet, as described above.

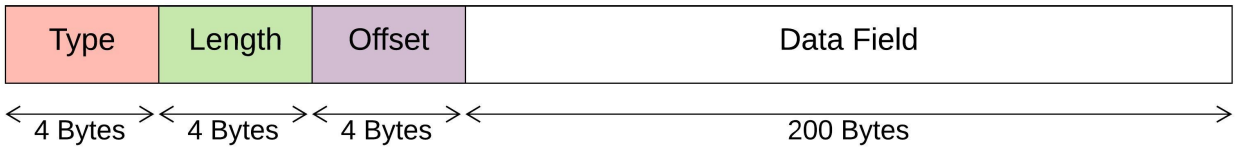

FIGURE 3.3: Data Packet

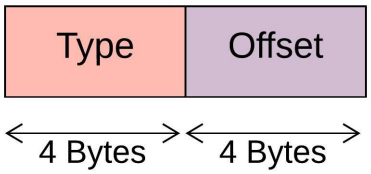

FIGURE 3.4: Offset Packet

3.3 Data and Offset Packets

Data Packet is the packet which contains the data being transferred from the Server to the Client (GET) and a Client to the Server (PUT).

Here is a description of what the different fields hold and mean :

Type - It holds the PacketType::DATA which distinguishes it from other packets and intimates the receiver of which packet it is receiving.

Length - It holds an Integer Value which is the actual size of the Data the sender has sent.

Offset - It holds an Integer Value which is the offset from which the Data has been produced.

Data Field - Holds the actual produced data.

Offset Packet is the packet which flows complementary to the Data Packet's flow. It is the packet which not only acts as the ACK for the previously received packets on the consumption side, but also provides the producer with the next offset to send data from.

Here is a description of what the different fields hold and mean :

Type - It holds the PacketType::ACK which distinguishes it from other packets and intimates the receiver of which packet it is receiving.

Offset - It holds an Integer Value which is the offset from which the Data is being requested. This also confirms that the data before this offset has been safely consumed.

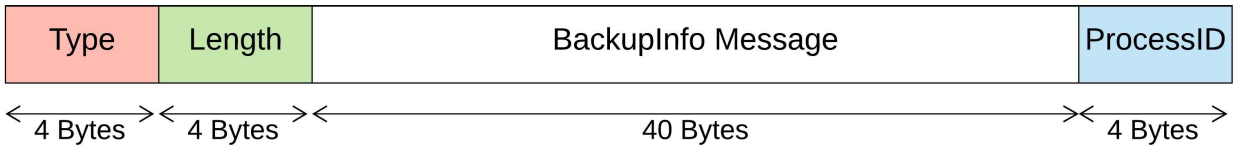

FIGURE 3.5: BackupInfo Packet

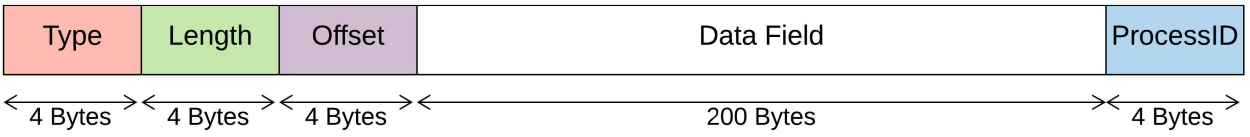

FIGURE 3.6: BackupData Packet

3.4 Backup Packets

Backup Packets are as the prefix suggests the packets which are send to the BACKUP Nodes. They are as can be seen very similar to the base packets with an extra payload of ProcessID. This ProcessID is very important to identify the Connection from which this packet has been received since the BACKUP Nodes receive Backup Packets from all Connections simultaneously.

BackupInfo Packet is the packet which a Data Channel sends the BACKUP Node before sending it Data for Backup. Here is a description of what the different fields hold and mean :

Type - It holds the PacketType::BACKUPINFO for distinction from other packets.

Length - It holds an Integer Value which is the actual size of the Info Message.

Info Message - Holds the Filename whose backup will be sent.

ProcessID - Holds the ProcessID of the Connection Channel.

BackupData Packet is the packet which a Data Channel sends the BACKUP Node containing the Data it has received from the PUT Client. Here is a description of what the different fields hold and mean :

Type - It holds the PacketType::BACKUPDATA for distinction from other packets.

Length - It holds an Integer Value which is the actual size of the Data sent for backup.

Data Field - Holds the data which needs to be backed up.

ProcessID - Holds the ProcessID of the Connection Channel.

Chapter 4

Deployment and Testing

Description

Deploying this Server-Client System on a real network would be a challenging task so we tested in on a Network Emulator Platform - Mininet.

4.1 Mininet and MiniNAM

Mininet is a Network Emulator platform that creates a network of virtual hosts switches and controllers within a single system. It is perhaps more precisely a network emulation orchestration system. It runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualisation to make a single system look like a complete network, running the same kernel, system, and user code. It provide each process(host) a separate network interface, such that it seems that the packets are being sent through a real Ethernet interface. Mininet's virtual hosts, switches, links, and controllers are the real thing – they are just created using software rather than hardware – and for the most part their behaviour is similar to discrete hardware elements. Since it is an emulator it is the best possible option to test the system as it actually uses the Network Protocol Stack, unlike Simulators.

MiniNAM [Khalid et al. (2017)] is a GUI Client that uses Mininet in its backend and does real-time animation over it using tkinter-imaging to visualise the packet flow going on in the emulated network.

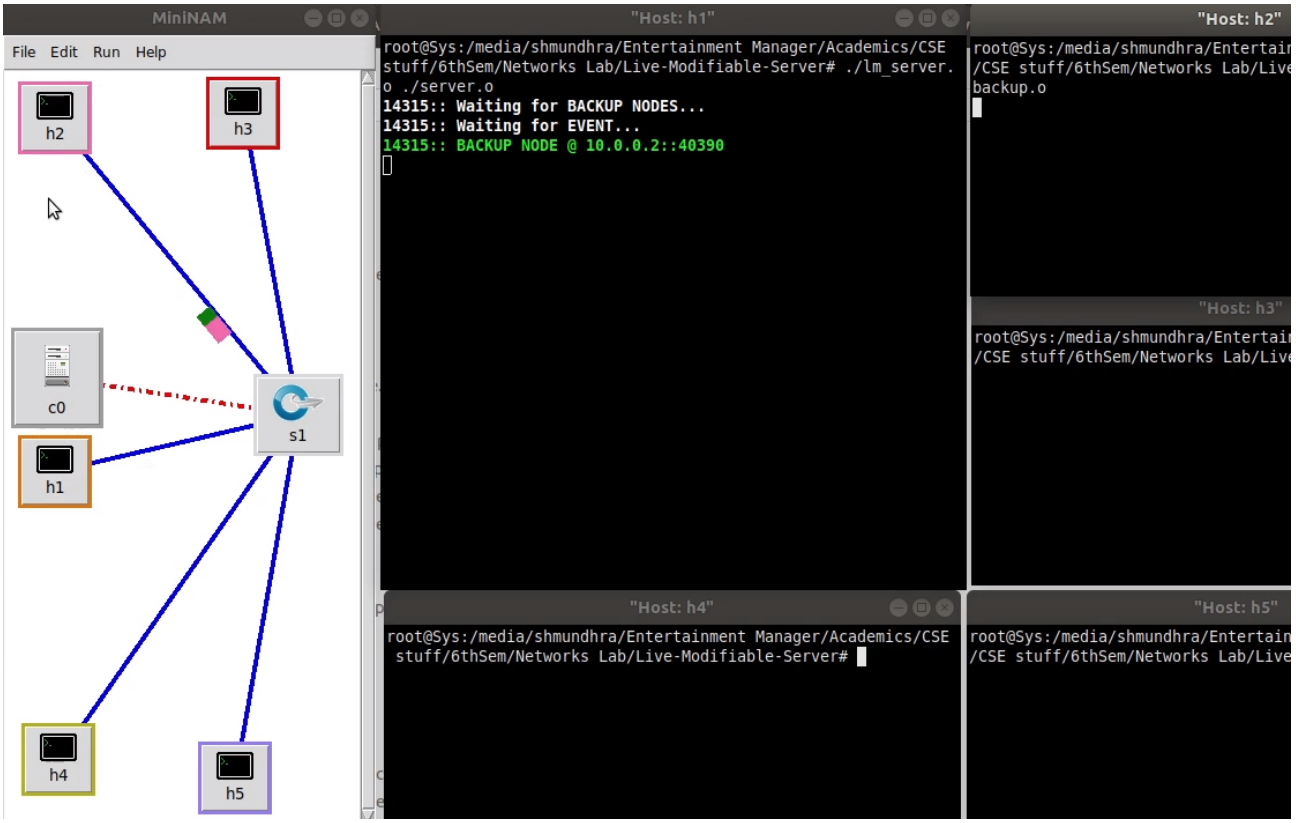

FIGURE 4.1: BACKUP Node Initiating Connection

4.2 Demo and Testing

The deployement and testing of the system was carried out on Mininet - [VIDEO LINK]

Control Channel accepting BACKUP Nodes As visible from Figure 4.1, the Control Channel has been spawned by the System Admin at Host-H1 and is waiting for new connections (with ProcessID - 14315). Now, when the Host-H2 attempts to register itself as a BACKUP Node at the Server, we see a Packet flowing from H2 to the switch, which is basically the TCP Connect request which is followed by a 3-way Handshake. Thereafter the BACKUP Node at 10.0.0.2::40390 is registered at the Server successfully.

All BACKUP Nodes Registered As visible from Figure 4.2, the Control Channel has accepted connection requests from two BACKUP Nodes at Host-H2 and Host-H3. They have been successfully registered at the Server side, and the confirmations have been sent to them in form of Info Packets which they have received. Now the System Admin has sent an EOF to the Control Channel which has made it come out of the loop for accepting BACKUP Nodes and it is now awaiting connections from Data Clients.

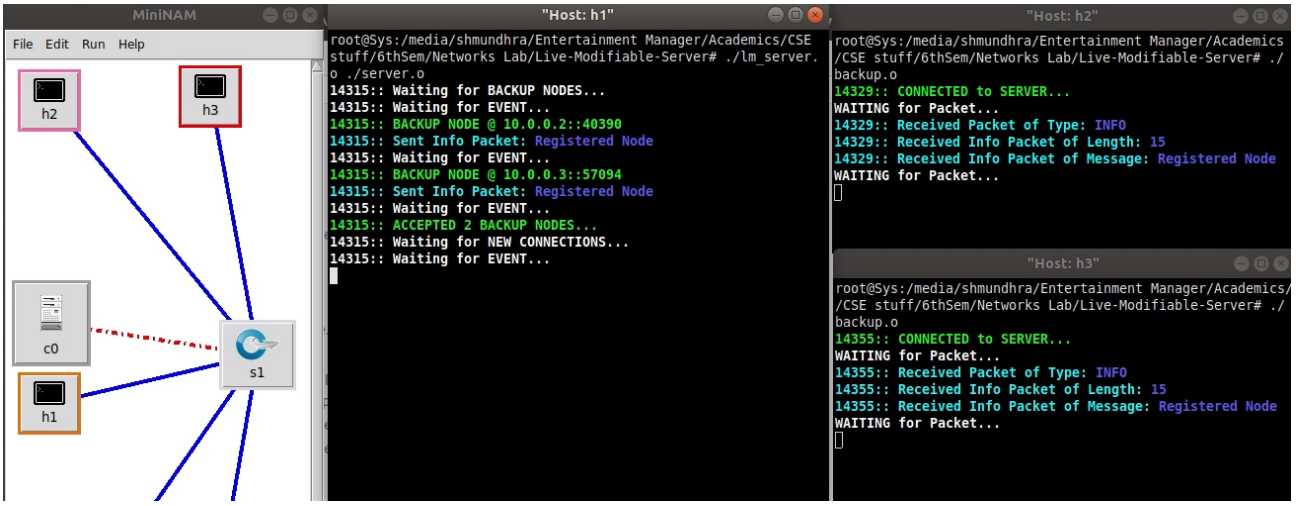

FIGURE 4.2: BACKUP Node Connections Complete

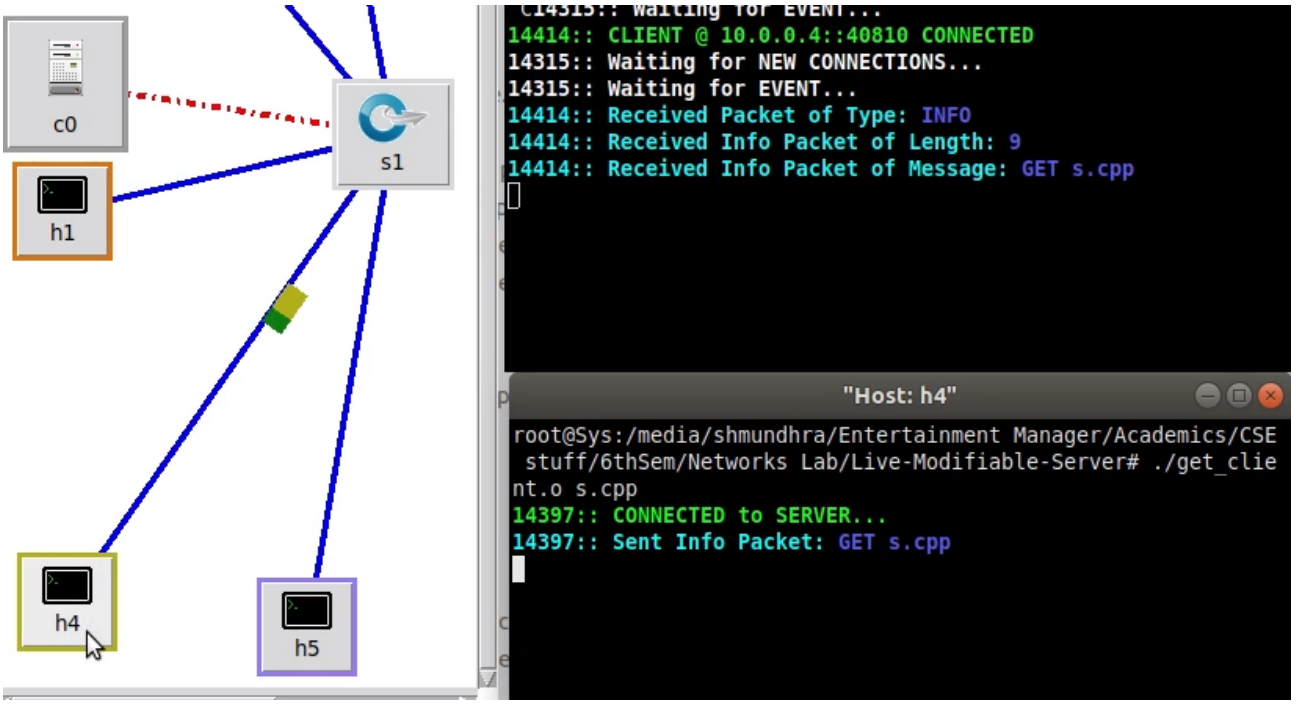

FIGURE 4.3: Connection of a GET Client

GET Client Connecting to Server A GET Client is started at Host-H4 which initiates connection at the Server (Figure 4.3). The Server registers a connection from 10.0.0.4::40810 and then forks a Connection Channel with ProcessID - 14414 to deal with this GET Client.

At the instant captured in the figure, the GET Client at H4 is sending an Info Packet with the Command - [GET s.cpp] to the Server at H1, thereby requesting the data of *s.cpp*.

GET Client and Code Modification Figure 4.4 and 4.5 demonstrates what happens when we pause a ongoing data transfer, here with the GET Client at H4 and then resume.

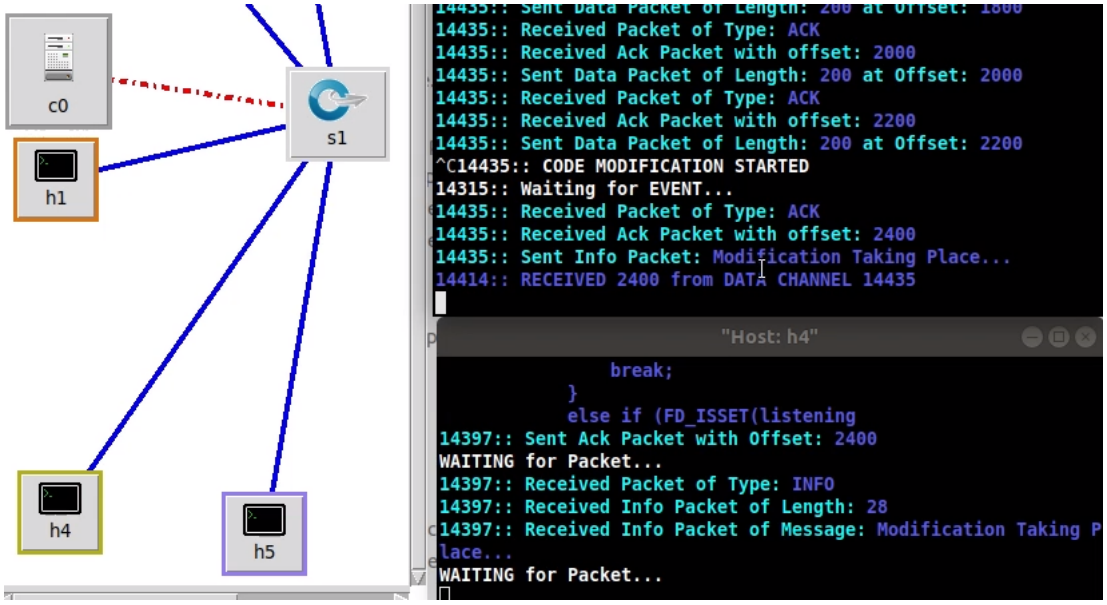

FIGURE 4.4: Pausing a GET Client

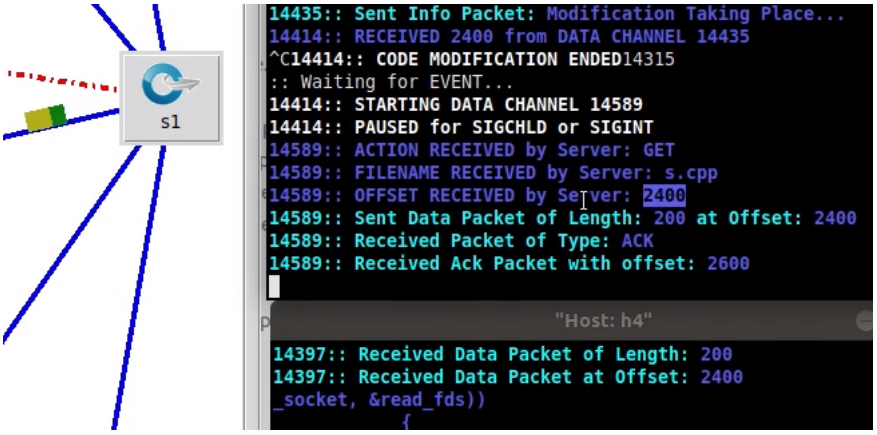

FIGURE 4.5: Resuming a Paused GET Client

Firstly we observe that the data transfer is taking place from the Data Channel at Server Side with ProcessID - 14435, different from the Control Channel and the Connection Channel. Anyway, so data packets are being sent from the Data Channel to the Client and as a response Ack Packets are being sent from the GET Client to the Data Channel.

Now a SIGINT is sent to the Process Group by the System Admin. This causes a SIGMODIFY to be sent to the Data Channel which signals it to end the ongoing transfer. The Data Channel then sends out an Info Packet to the GET Client intimating it about the code modification taking place at the Server. It further sends the Offset - 2400 till which data has successfully been consumed at the Client, to the Connection Channel (PID 14414). After resumption by another SIGINT to the Connection Channel, a new Data Channel (PID 14589) is spawned and it starts data transfer from the previously received offset - 2400.

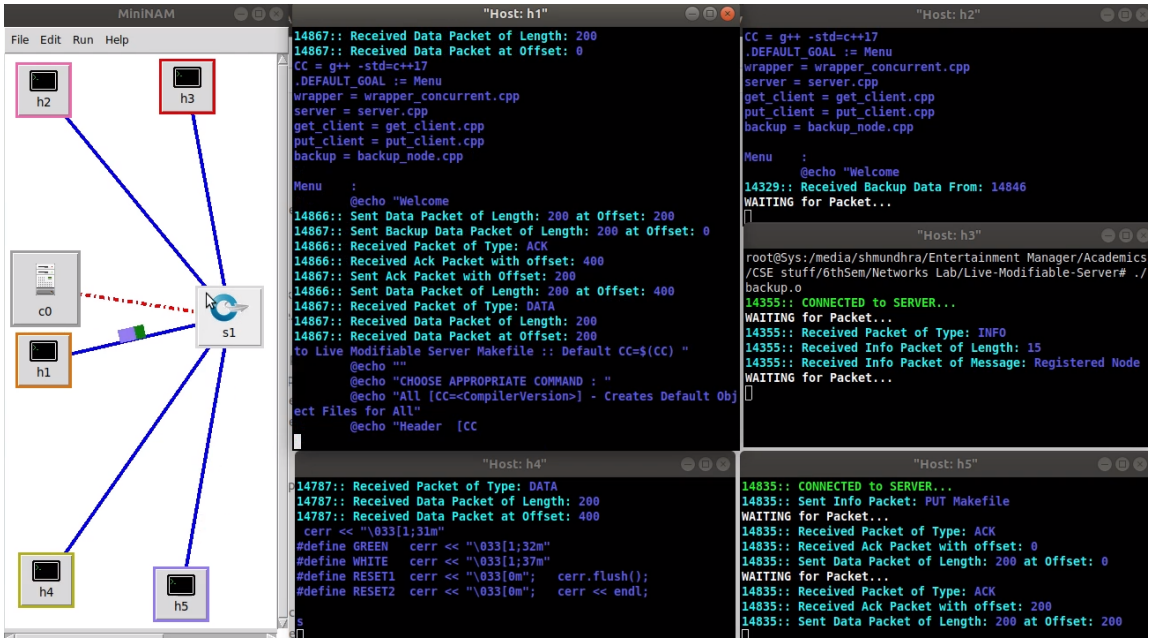

FIGURE 4.6: PUT Client Sending Data and BACKUP Node H2 receiving the Backup

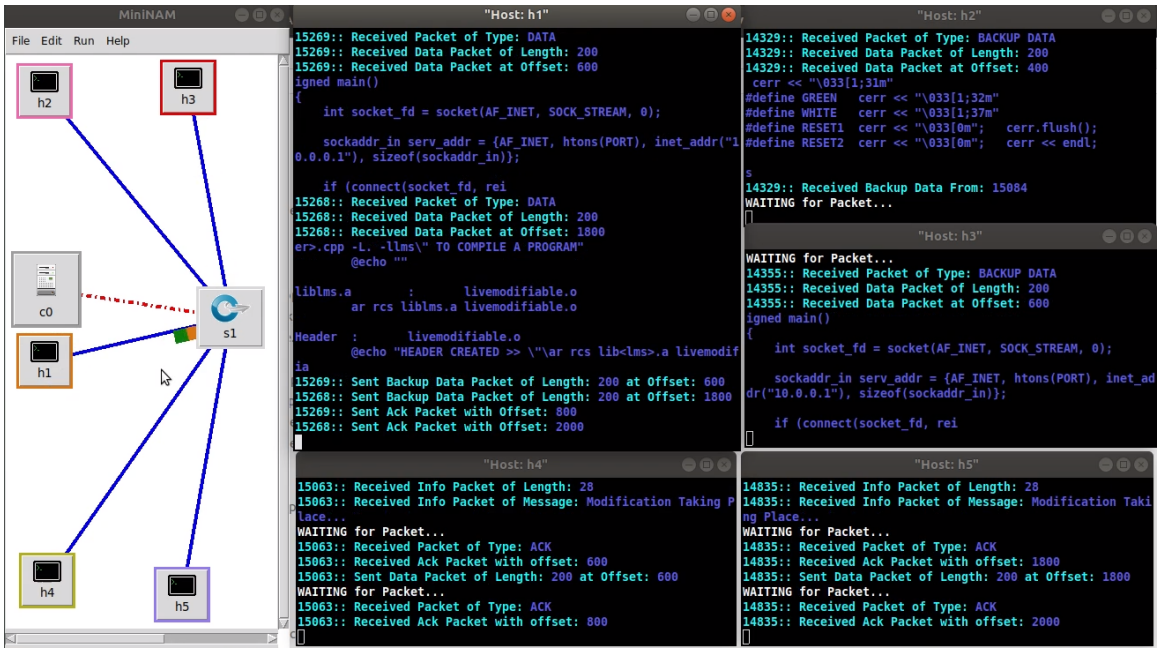

FIGURE 4.7: PUT Client Sending Data and BACKUP Node H3 receiving the Backup

PUT Client and BACKUP Nodes In Figure 4.6 we see that Host-H5 has connected to the Server as a PUT Client. The Host-H2 is the current BACKUP Node and is receiving the data sent from the H5 to H1. *Notice the similarity in printed data on top of terminal H1 and H2.* After code modification we can see in Figure 4.7 that now Host-H3 is the BACKUP Node in use and receives the BackupData. *Notice the similarity in printed data on top of terminal H1 and H3 this time.*

Chapter 5

Conclusion

There are a number of maintenance and update tasks which need to be performed on a network at a regular basis. Upgrading of the firewall running on a Virtual Machine or a Container, Modifying the logic of the Proxy Load Balancer, Reallocating bandwidth to different connections by changing the notion of fairness - basically changing the logic of certain network functions. The cross dependency of these functions poses a major problem in isolating and changing them individually.

Nowadays there is a gradual shift and people are switching the Virtual Network Functions because they provide a more granular access to the network functionalities. Even 5G and 6G Networks are thinking of putting all the cellular network functionalities as Virtual Network Functions. The main reason for this is to be give programmability to the Network Service Providers. If the Network Service Provider tries to make a change in the VNF, they have to suspend the service for those users who are currently using that VNF.

The problem of downtime however while modification occurs still exists, and this is where we shall come in.

Bibliography

Khalid, A., Quinlan, J. J., and Sreenan, C. J. (2017). Mininam: A network animator for visualizing real-time packet flows in mininet. In *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*, pages 229–231.