



# **LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT**

**Pseudo System Protocol for Information Transfer**

**UE18CS390B – Capstone Project Phase – 2**

*Submitted by:*

<b>Akshay Vasudeva Rao</b>	<b>PES1201800310</b>
<b>Amogh R K</b>	<b>PES1201801309</b>
<b>Prajwal K Naik</b>	<b>PES1201801455</b>
<b>Rohan Iyengar</b>	<b>PES1201800547</b>

Under the guidance of

**Prof. Pushpa G**  
Assistant Professor  
PES University

**August - December 2021**

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING  
FACULTY OF ENGINEERING  
PES UNIVERSITY**

(Established under Karnataka Act No. 16 of 2013)

**TABLE OF CONTENTS**

1. Introduction	4
1.1 Overview	4
1.2 Purpose	4
1.3 Scope	4
2. Proposed Methodology / Approach	9
4.1 Algorithm and Pseudocode	9
4.2 Implementation and Results	9
4.3 Further Exploration Plans and Timelines	9
Appendix A: Definitions, Acronyms, and Abbreviations	9
Appendix B: References	9
Appendix C: Record of Change History	9

**Note:**

<b>Section 1</b>	<b>Common for Prototype/Product Based and Research Projects</b>
<b>Section 2</b>	<b>Applicable for Research Projects.</b>
<b>Appendix</b>	<b>Provide details appropriately</b>

**1. ABSTRACT Introduction**

**1.1. Overview**

Communication protocols have always looked at data security and host security as unrelated goals. Protocols like TLS have been successful in ensuring data security. The goal of PSPIT is to take a unified approach to data and host security by implementing pseudo-systems that help hosts isolate.

PSPIT aims to ensure host and data security while limiting latency to a minimum with the additional constraint of making the implemented pseudo-systems as economic as possible (by limiting buffer size and computational ability to the bare minimum).

The study began with an in-depth review of existing literature on TCP, SSL, and TLS which ultimately form the bedrock upon which PSPIT is built. It then moves on to a description of the practicalities of the implementation of the first proposed prototype - a socket simulation of the protocol.

This low-level design document describes the protocol in detail in a theoretical and conceptual manner, from where the focus shifts to the actual implementation.

## **1.2. Purpose**

If one draws a parallelism between a computer and a building, one might say the ports behave as doors that facilitate the movement of people, who serve as metaphors for packets of data. Networking protocols would-be managers who'd decide what kind of people have admitted entry (packet formats), whom to expect when (packet exchange sequences) and when to open the doors, and more importantly, for whom (host authentication). Firewalls would then serve as guards while anti-virus systems would be metaphoric bouncers, identifying and ejecting suspected troublesome people (malware).

This setup is far from foolproof and has an inherent set of flows. The safety of the building (the host) rests on the efficacy of the managers, guards, and bouncers in performing their respective duties. Even if they performed their jobs diligently, there's always the risk of malicious people finding new and creative ways to enter the establishment either by masquerading their intentions (trojans) or by simply finding alternate doors with possibly less protection.

Now, imagine the building's address was secret. In this scenario, all potential entrants (packets) were received at a holding area disconnected from the main building where they would be thoroughly inspected and any suspicious persons would be ejected. The people who pass the inspection would then be securely transported to the main building (host) in a way that prevents them from ever knowing its address. In this way, the building is much more inherently protected as it remains virtually nonexistent to attackers.

The Pseudo-System Protocol for Information Transfer uses pseudo-clients and pseudo-servers to enable hosts to communicate through the internet while being completely isolated (offline) from the internet. It enables hosts to use the power of the

internet while making the most of the safety of remaining offline and isolated, providing users with the best of both worlds.

### **1.3. Scope**

Most commercially used communication protocols enable only data security. PSPIT goes a step further and implements host security by essentially allowing communication via the internet by never connecting to it.

This prototype enables hosts to use the power of the internet while making the most of the safety of remaining offline and isolated, providing users with the best of both worlds.

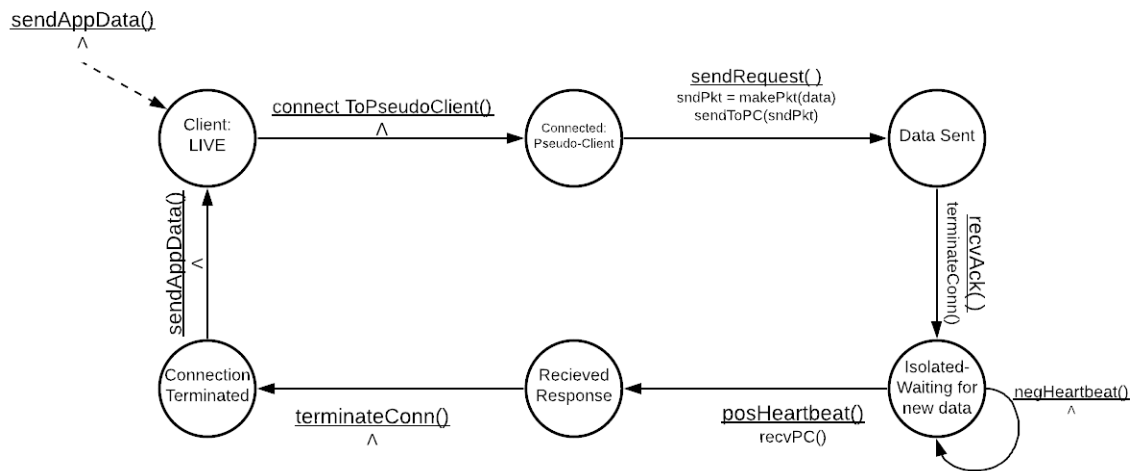
This protocol secures the data being transferred and masks the host so that a third party cannot intervene for stealing data or gain access to the hosts. The protocol sequences the transfer of data such that the hosts always remain disconnected from the internet while using the internet as the primary medium for the transfer, thus making full use of the power of the internet but without any of the risks of intrusion that come with it.

Today's protocols do not contain an inclusion for end system isolation, and PSPIT allows for information to be sent and received without the end systems ever being directly connected to the internet, which significantly reduces the risk of attack.

## 2. Proposed Methodology / Approach

### 2.1 State Diagrams

#### a. Client



This is the state diagram for the client/host system. There are 6 states including that of the terminal isolated state. The client system triggers the entire protocol when it receives a send command from the upper layer protocol. The first step in this cycle is the establishment of a connection with the pseudo-client. This is done using standard TCP procedures. Once the connection is established (and secured) the client sends the request packet through the data flow link to the pseudo-client. This is again handled by the underlying TCP which takes care of the delayed or unsuccessful transmission. Upon receipt of acknowledgment of successful transmission of the request packet, the client terminates its connection with the pseudo-client. At this point, the client is isolated from the pseudo-client and hence from the internet.

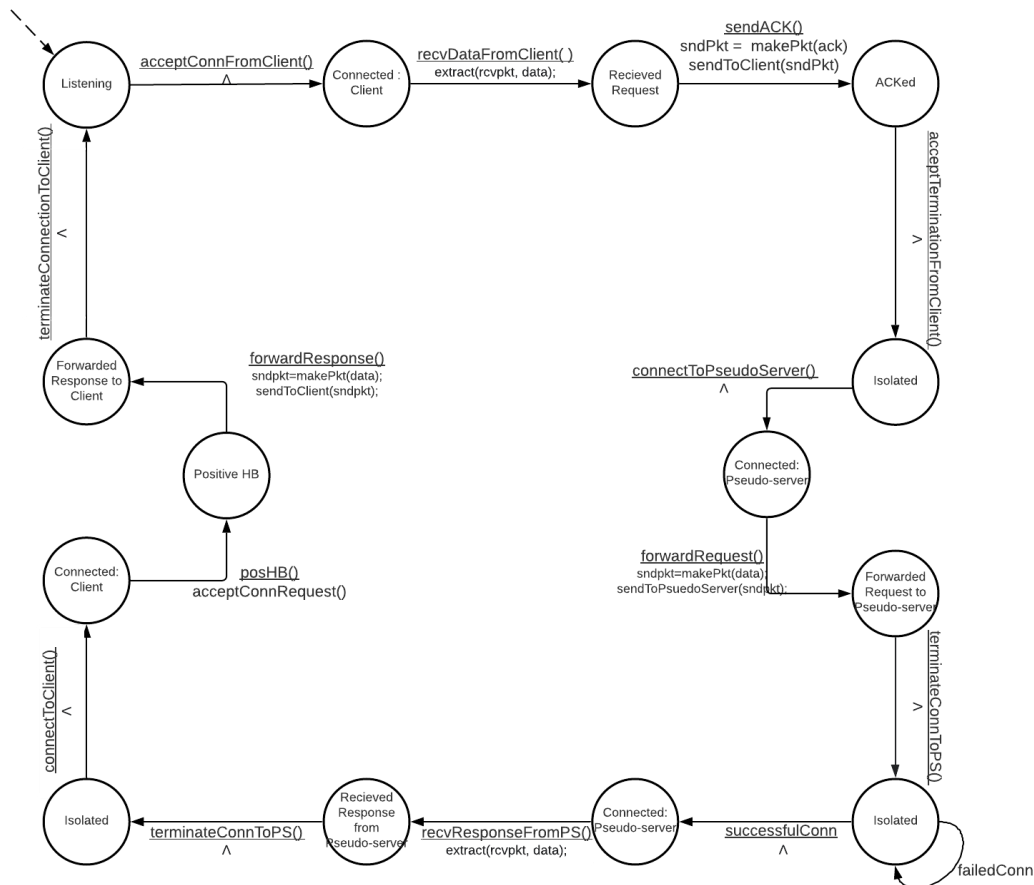
Immediately after isolation of the client, the heartbeat phase of the client takes over. The client sends repetitive beacons in order to check for the availability of response to its request at the pseudo-client. This is done using UDP in order to reduce the latency

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

of the entire protocol. Since the connection with the pseudo-client is closed, the client is unable to send the beacons (this connection is later established by the pseudo-client).

Once the heartbeat beacon returns, indicating that the pseudo-client has a new response to send to the client, the connection is re-established with the pseudo-client and the client receives the pending response. After this is done, the client terminates its connection with the pseudo-client and is thus isolated from the network.

### b. Pseudo-Client



This is the finite state diagram for the pseudo-client system of PSPIT. This system can be in 7 states in 2 directions throughout a request-response cycle. The initial state

## **LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT**

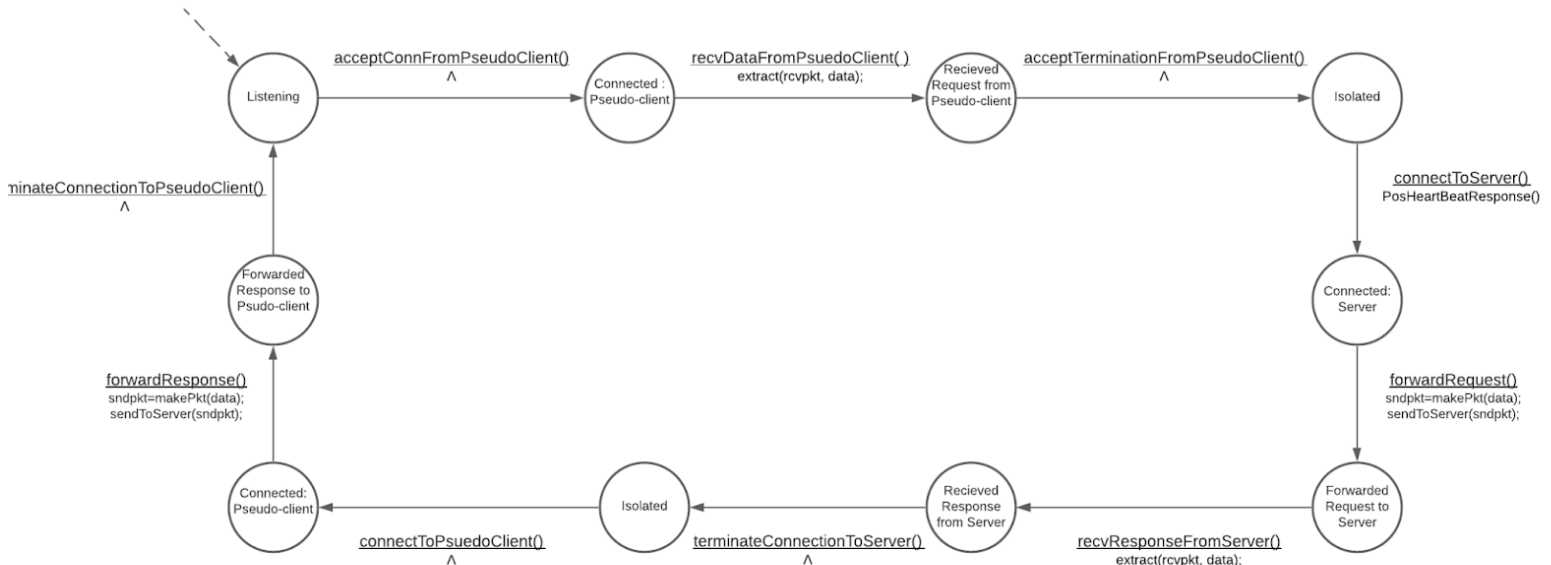
is achieved when the client system(depicted in the previous FSM) tries to establish a connection with the pseudo-client. The pseudo-client accepts this connection request. Once a connection is established using standard TCP procedures, the pseudo-client receives the request packet from the client. Upon successful receipt of this request, the system sends an ACK message as an acknowledgment. Once the client receives the ACK, the connection is terminated between itself and the pseudo-client. At this point in the protocol, the pseudo-client is isolated from the other systems as there are no active connections to or from itself.

As its next phase, the pseudo-client connects to the pseudo-server and forwards the request packet to the pseudo-server. Upon receiving an acknowledgment of successful transmission of the same, the pseudo-client terminates its connection to the pseudo-server. Immediately after this step, the pseudo-client starts attempting to establish another connection with the pseudo-server. Since the latter is not accepting connections at the moment, all these connection requests fail. When the pseudo-server has new response packets to send, the connection requests succeed and a new connection for receiving the response packet is established. After receiving the response packet from the pseudo-server, the connection to the pseudo-server is terminated. Note that the transmission of the packet is done using normal TCP procedures.

The next phase is to relay the newly received packet to the client. All this while, the client would have been sending heartbeats probing for the existence of a new response message. Once the pseudo-client is prepared to forward the response to the client, it accepts a connection request from the client and responds to the heartbeats sent by the client. After this, it transmits the response message hence completing its part in the protocol.



### c. Pseudo-Server



This is the finite state diagram for the pseudo-server system of PSPIT. The initial state for this machine is that it is in a listening and isolated state waiting for a connection request from a pseudo-client. From the previous FSM given for pseudo-client, we can see that once the connection is established between pseudo-client and pseudo-server, the data transmission takes place via TCP protocol procedure. Once the data packet is received, the pseudo-server terminates the connection and goes into an isolated state(no internet connection).

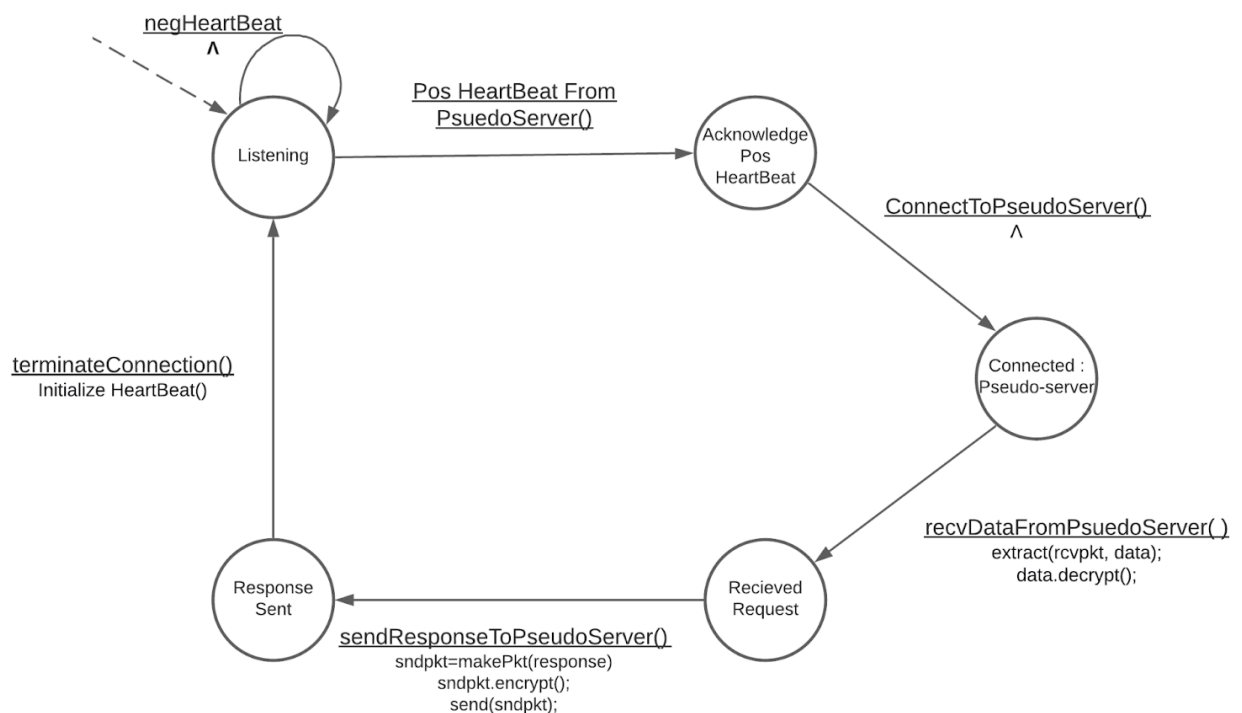
In the next phase, the pseudo-server keeps receiving heartbeats from the server, but only sends a positive response when it has data packets to be transmitted to the server and a connection with the server is established. The request is forwarded to the server using normal TCP procedures and the server responds back to the request. After receiving a response from the server, it terminates the connection with the server leading it to an isolated state.

And in this phase, the isolated pseudo-server acknowledges connection requests from the pseudo-client, as the response has been received from the server, via a normal

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

TCP three-way handshake connection. The response is forwarded to the pseudo-client and the connection with the pseudo-client is terminated. Leading it back to the first state where it is in an isolated state waiting for the pseudo-client to send the request to establish the connection.

### d.Server



This is the finite state diagram for the server system of PSPIT. This machine has 5 states including the listening/isolated state. This is the final machine to which all the data packets are being sent. In the initial state of this machine, the server is an isolated state, sending heartbeats to the pseudo-server. The server sends repetitive beacons in order to check for the availability of requests at the pseudo-server. This is done using UDP in order to reduce the latency of the entire protocol. Since the connection with

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

the pseudo-server is closed, the server is unable to send the beacons(this connection is later established by the pseudo-server).

In the next phase, the pseudo-server sends a positive response to those heartbeats, which are then acknowledged by the server and a connection is established. Request/Data packets are sent from the pseudo-server to the server and the server responds back with the necessary data packets/response requested by the client to the pseudo-server. Then it goes back to the isolated state, wherein it restarts sending heartbeats to the pseudo-server, and the cycle repeats itself.

### 2.2 Algorithm and Pseudocode

#### a. Client

Sending Request to Pseudo-Client	<pre> clientSocket = createSocket() clientSocket.connect(pseudoClient) pkt = createReqPacket() clientSocket.sendReq(pkt) ack = recvAck() clientSocket.close() </pre>
Receive Response from Pseudo-Client	<pre> //use socket returned from heartbeat module as clientSocket  serverResponse = clientSocket.recieve()  clientSocket.close() </pre>
Heartbeats	<pre> while(true): </pre>

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

	<pre> try:     clientSocket = createSocket(pseudoClient).connect()     hb = createPacket(heartBeat)     clientSocket.send(hb)     hb_reponse = clientSocket.receive()     if(hb = positive):         return clientSocket except:     if(clientSocket):         clientSocket.close() </pre>
--	--

### b. Pseudo-Client

Receiving Request from Client	<pre> fromClientSocket = createSocket(client) fromClientSocket.acceptConnection() request = fromClientSocket.receive() ack = makePacket("ACK") fromClientSocket.send(ack) fromClientSocket.close() </pre>
Forwarding to Pseudo-Server	<pre> pseudoServerSocket = createSocket(pseudoServer) //data received from client request = makePacket(data) pseudoServerSocket.send(request) pseudoServerSocket.close() while(true): </pre>

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

	<pre> try:     pseudoServerSocket = createSocket(pseudoServer)     break except:     pass     sleep(1) //This can be a moving average value </pre>
Receiving Response from Pseudo-Server	<pre> response = pseudoServerSocket.receive() pseudoServerSocket.close() </pre>
Forwarding Response to Client	<pre> respondHeartbeat(client)  //toClient socket obtained from respondHeartbeat module  toClientSocket.send(response) toClientSocket.close() </pre>
respondHeartbeat	<pre> toClientSocket = createSocket(client) toClientSocket.connect() hb = recieveHB() toClientSocket.send("NEW") return toClientSocket </pre>

**c. Pseudo-Server**

Receiving Request from Pseudo-Client	<pre> pseudoClientSocket = createSocket(pseudoClient) request = pseudoClientSocket.receive() ack = makePacket("ACK") pseudoClientSocket.send(ack) pseudoClientSocket.close() </pre>
Forwarding Request to Server	<pre> respondHeartBeat() //serverSocket returned by respondHeartBeat module request = makePacket(request) serverSocket.send(request) </pre>
Receive Response to Server	<pre> response = serverSocket.receive() serverSocket.close() </pre>
Forward Response to Pseudo-Client	<pre> pseudoClientSocket = makeSocket(pseudoClient) pseudoClientSocket.acceptConnReq() response = makePacket(response) pseudoClientSocket.send(response) pseudoClientSocket.close() </pre>
Respond Heart Beat	<pre> serverSocket = createSocket(server) </pre>

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

	<pre> hb = serverSocket.receive() hb_res = makePacket("NEW") serverSocket.send(hb_res) return serverSocket </pre>
--	---

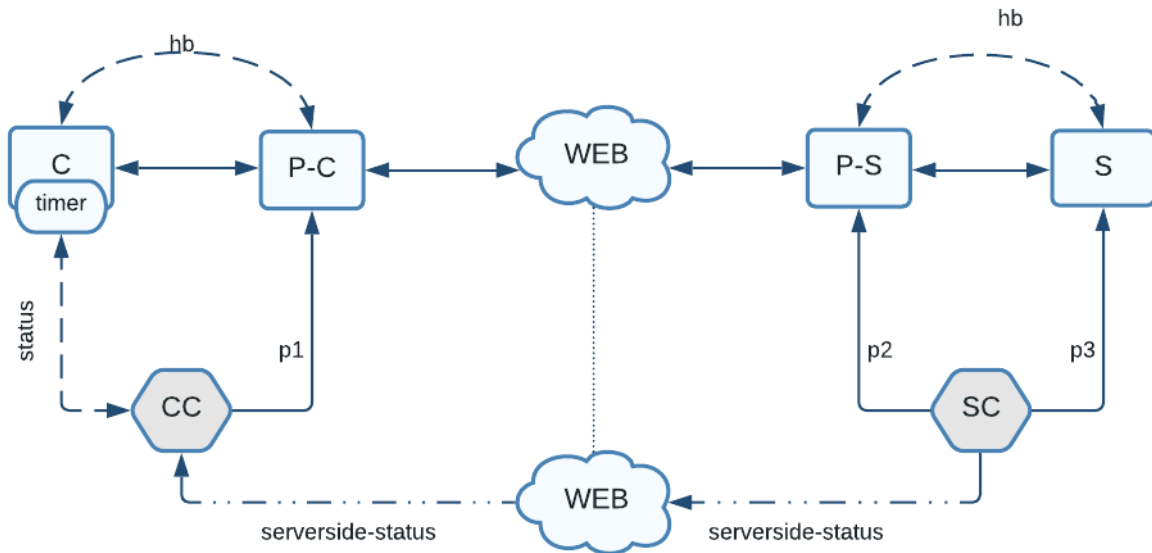
### d. Server

Receive Request from Pseudo-Server	<pre> //pseudoServerSocket received from heartbeat module  request = pseudoServerSocket.receive() </pre>
Respond to Request	<pre> response = makePacket(reponse) pseudoServer.send(response) pseudoServer.close() </pre>
Heartbeats	<pre> while(true):     try:         pseudoServerSocket = createSocket(pseudoServer)         hb = makePacket(hb)         pseudoServerSocket.send(hb)         hb_response = pseudoServer.receive()         if(hb_reponse == "NEW"):             return pseudoServerSocket     except:         if(pseudoServerSocket):             pseudoServerSocket.close() </pre>

### 2.3 System-Failure Handling

The pseudo-system protocol for information transfer despite enforcing stringent host security measures is prone to failures or nodes or delays of request-response messages like any other protocol. After all, the nodes in the system are physical machines running on sources of power with the possibility of being miles apart from each other which can contribute towards packet delays and losses. While delays are already handled by lower layer protocols like TCP, there are a few methods to implement delay detection/handling. Additionally, there are multiple methods to handle system downtimes/failures.

#### 2.3.1. System Check Nodes



Here, two extra nodes(routers) are added to the entire topology. These are the Client Check(CC) on the client side and Server Check(SC) on the server side. These nodes maintain the availability of the entities in the protocol. Server Check handles the availability of the two server side machines, namely the pseudo-server and the server machine and Client Check handles the availability of the client side node pseudo-client.



## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

Additionally, Client Check maintains the status of the server side machines in the form of bit strings. Eg, 101 means that the pseudo-client and the server are active but the pseudo-server system is down.

Client Check sends probe p1 to pseudo-client while Server Check sends probe p2 and p3 to pseudo-server and server, respectively. These probes are sent through a TCP channel which is initially set up when the systems come online for the first time. So, these channels are up and running as long as both sides of these systems are up and running. This means that at any point of time, Server Check is able to send a probe to check the availability of the systems under its supervision. The same also applies to Client Check.

The moment Server Check is unable to send a probe to its child systems, it implicitly assumes that the system it was trying to probe is down. In the occurrence of such an event, the Server Check sends a `SYSTEM_DOWN` status message to the Client Check through the world wide web using conventional TCP protocols. The `SYSTEM_DOWN` message is simply a bitstring showing the availability of the systems under the Server Check node supervision. The next time (given that a `SYSTEM_DOWN` message has been sent) it is able to send the probes successfully to both the server side systems, it sends a `SYSTEM_UP` message to the Client Check machine.

Similarly, the Client Check continually probes the liveness of the pseudo-client system by probing it at regular intervals. Upon encountering the failure of the pseudo-client, it internally sets the status bit of the pseudo-client machine to 0. The next time it is able to probe the pseudo-client successfully, it resets the bit for that particular system.

Upon receiving the `SYSTEM_DOWN` message from Server Check, it changes the status of the respective machines to 0.

This is the internal workings of Server and Client Check machines.

Every request sent by the client starts a timer exclusive to that request. The expiration of the timer could imply the occurrence of two events.

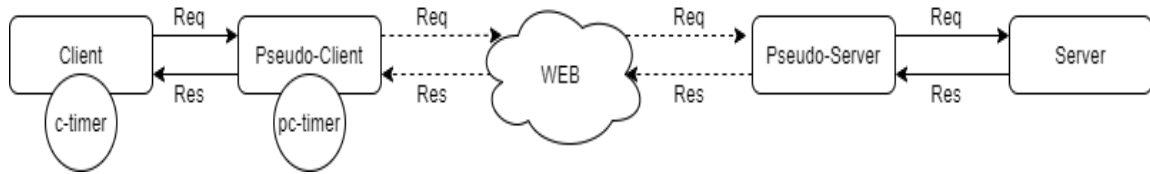
## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

- Delay in receiving the response: In this case, all systems taking part in the protocol are live. However, the excess traffic in the external internet is causing a delay in the transmission of packets in the lower layers. To detect this, the client sends a STATUS\_CHECK message to Client Check. This results in a response with a code of 111 which implies that all the machines in the protocol are live and functioning. The client then restarts the timer for that particular request. Upon three such cycles of timer + live discovery, the client immediately retransmits the request message.
- Failure of the systems: In this case, one of the systems in the protocol has failed. This is almost always preceded by the receipt of the SYSTEM\_DOWN message by the Client Check. Upon expiry of the timer in the client for a particular request, it sends a STATUS\_CHECK message to the Client Check. This results in a response with an indication that one of the systems is down. Eg, 101, 001, 100 etc. This is followed by the immediate re-transmission of the request by the client.

Drawbacks: The extra nodes added to the protocol, Client Check and Server Check are also additional power dependent machines implemented to keep track of the status of the machines taking part in the protocol. These are also susceptible to the same probability of failure as that of the protocol systems. If there is a failure of the check machines, the client would have to assume that one of the protocol systems is dysfunctional and wait for it to come online. However, this system does solve the task of identifying the faulty systems.

Additionally, these systems could be compromised and false data could enter the client machine leading to security lapses. A possible solution to this problem is to ensure that the check machines are isolated before flow of data to the client.

### 2.3.2. Dedicated Timers in Each System



In this proposed solution, each of the client-side machines have a dedicated timer built into each of them. These timers are asynchronous with respect to each other. Additionally, these timers are always running which means that, the moment each of those systems start up, the timer associated with that system is activated. The timers are depicted in the above diagram as follows:

- c-timer: This is the client timer exclusive to the client machine.
- pc-timer: This is the pseudo-client timer exclusive to the pseudo-client machine.

These timers are responsible for only the requests leaving their respective machines and do not keep track of the delay/latency of the requests handled by the other machines.

When the client system sends a request to the pseudo-client, its timer is started. This timer ends only when the response for that particular request is received back at the client. When the timer expires, the client resends the request message and restarts the timer. In case the client receives a delay indicator message from the pseudo-client before the expiry of its own timer, it ends the timer immediately and resends the request to the pseudo-client. In case, the client is unable to establish a connection with the pseudo-client initially, it implies that the latter is dysfunctional and is unable to receive requests at the moment. However, the problem with using this methodology is that the client attempts to send a heartbeat immediately after sending a request message as well, and the inability to send the heartbeat could be mistaken for the dysfunctionality of the pseudo-client. In this case, the timer of the client is used as a fall back. The moment the timer expires, the client assumes that there is a delay or one of the systems has failed.

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

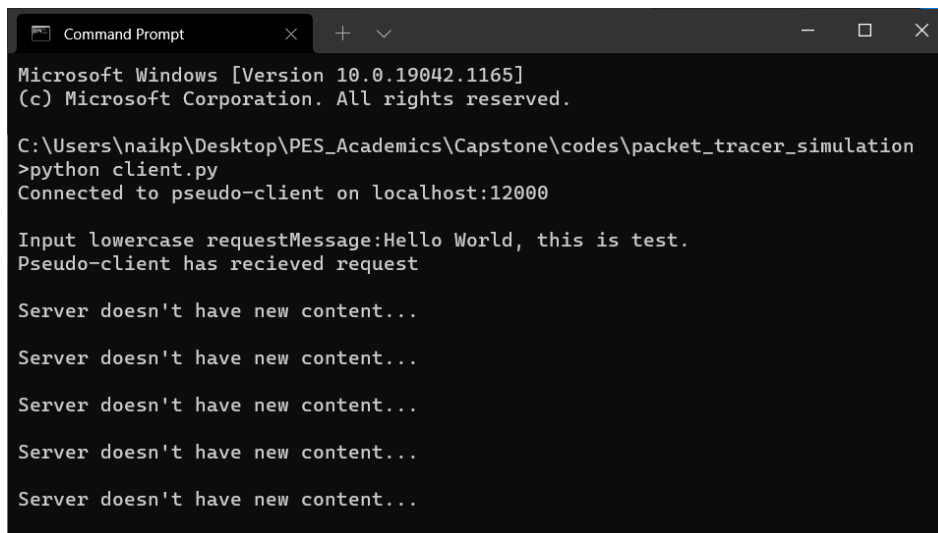
When the pseudo-client system sends a request to the pseudo-server, its timer is started; this timer ends only when the response for that particular request is received back at the pseudo-client. When the timer expires, the pseudo-client relays a delay indication message to the client. The client can then resend the request. In case the pseudo-client receives a delayed indicator message from the pseudo-server before its timer expires, it immediately ends its timer and relays this delay message to the client. Again, the client can attempt to resend the message. If the pseudo-client is unable to send the request in the first place, it can be implied that the pseudo-server or the server is disabled and a delay indicator message is sent to the pseudo-client. However, since the pseudo-client utilizes the inability to connect to the pseudo-server as a means to identify if there are new responses at the pseudo-server, this could cause havoc. The solution to this is that, if the request is sent and the pseudo-client is unable to connect to the pseudo-server, it is assumed that the server side machines are performing correctly. In case this assumption turns out to be incorrect, the already running timer for that request eventually expires and the pseudo-client can then send a delay indicator to the client.

The pseudo-server also has a fall back mechanism in case of errors. In case the pseudo-server is unable to send the request to the server, it can send a delay indicator to the pseudo-client which is relayed upstream to the client which can then resend the request once the server is online.

**Drawbacks:** The major drawback of this solution is that it is not easy to decide if the timeout is due to the delay in transmission of the messages or due to the failure of the machines.

## 2.4 Implementation and Results

### 2.3.1 Prototype in Action



```
Microsoft Windows [Version 10.0.19042.1165]
(c) Microsoft Corporation. All rights reserved.

C:\Users\naikp\Desktop\PES_Academics\Capstone\codes\packet_tracer_simulation
>python client.py
Connected to pseudo-client on localhost:12000

Input lowercase requestMessage:Hello World, this is test.
Pseudo-client has recieved request

Server doesn't have new content...

Server doesn't have new content...

Server doesn't have new content...

Server doesn't have new content...

Server doesn't have new content...
```

Presently, the functioning of the protocol is illustrated using 4 terminals to simulate the 4 systems in the protocol: client, pseudo-client, pseudo-server and server.

The prototype is initiated by running in the following order.

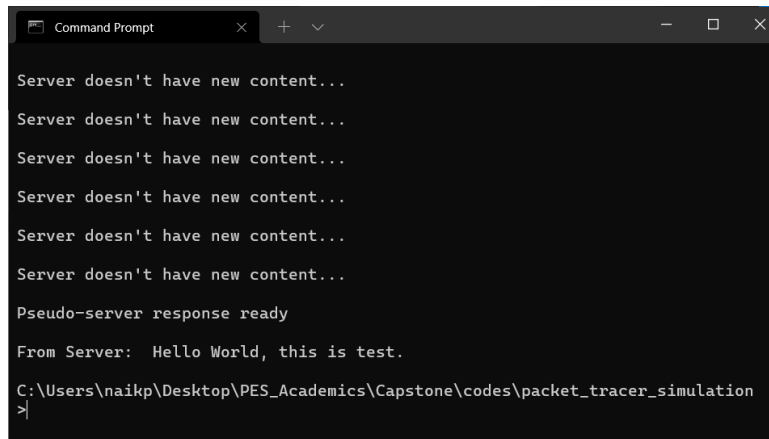
- Server
- Pseudo-server
- Pseudo-client
- Client

The above screenshot belongs to the simulated client terminal.

This is the result obtained after executing the client protocol script. In the present prototype, the client requests for input from the user. Immediately after this phase is done, the client starts probing the pseudo-client for the presence of a new response from the server.

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

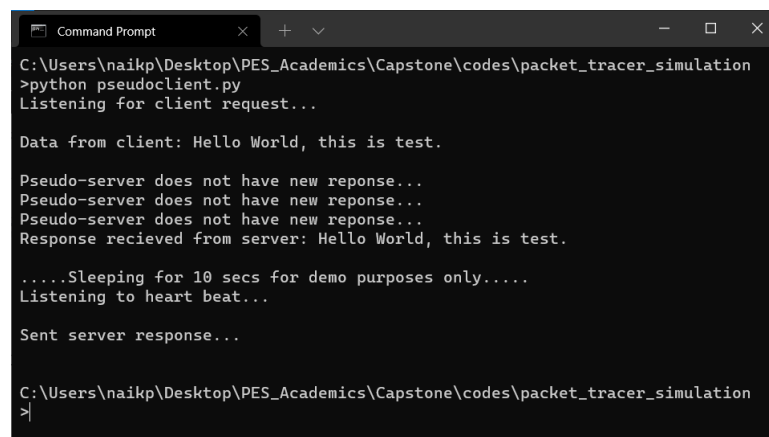
The consecutive “Server doesn’t have new content” messages are the outputs that indicate that the pseudo-client does not have new responses. This is done by attempting to send a packet through an open socket on the client.



```
Command Prompt
Server doesn't have new content...
Server doesn't have new content...
Server doesn't have new content...
Server doesn't have new content...
Server doesn't have new content...
Server doesn't have new content...
Pseudo-server response ready
From Server: Hello World, this is test.
C:\Users\naikp\Desktop\PES_Academics\Capstone\codes\packet_tracer_simulation
>
```

As long as the pseudo-client has no response to provide to the client, it continues to probe the pseudo-client. The moment the pseudo-client has a new response, it responds to the probe. This step is indicated by the “Pseudo-server response ready” message being displayed on the client terminal

Once this is done, the pseudo-client sends the server response to the client. This is indicated by the message “From Server: Hello, World, this is test”.



```
Command Prompt
C:\Users\naikp\Desktop\PES_Academics\Capstone\codes\packet_tracer_simulation
>python pseudoclient.py
Listening for client request...

Data from client: Hello World, this is test.

Pseudo-server does not have new reponse...
Pseudo-server does not have new reponse...
Pseudo-server does not have new reponse...
Response recieved from server: Hello World, this is test.

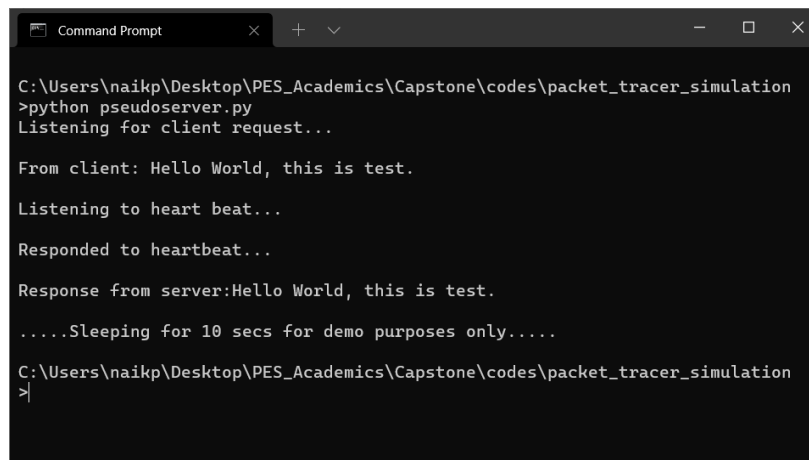
.....Sleeping for 10 secs for demo purposes only.....
Listening to heart beat...

Sent server response...

C:\Users\naikp\Desktop\PES_Academics\Capstone\codes\packet_tracer_simulation
>
```

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

This is the pseudo-client screenshot. Initially, it receives the request from the client. This is indicated by “Data from client: Hello World, this is test”. Immediately after that, it closes the client port, forwards the request to the pseudo-server, closes the pseudo-server port and tries to re-establish the connection with the pseudo-server. The re-establishment is indicated by “Pseudo-server does not have new response”. Once the connection has been re-established, the pseudo-client receives the response and forwards it to the client. This is indicated by “Response received from server: Hello World, this is test” and “Sent server response”



```
Command Prompt
C:\Users\naikp\Desktop\PES_Academics\Capstone\codes\packet_tracer_simulation
>python pseudoserver.py
Listening for client request...

From client: Hello World, this is test.

Listening to heart beat...

Responded to heartbeat...

Response from server:Hello World, this is test.

....Sleeping for 10 secs for demo purposes only....

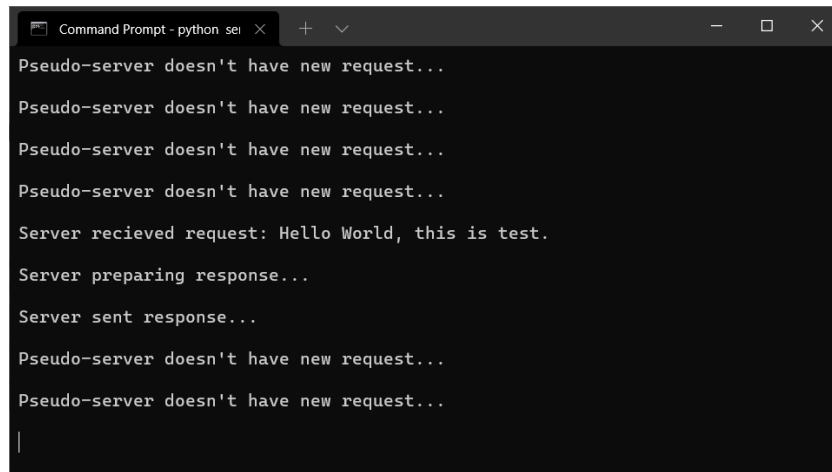
C:\Users\naikp\Desktop\PES_Academics\Capstone\codes\packet_tracer_simulation
>|
```

This terminal initiates the pseudo-server python file. After the initiation, it starts listening for requests from pseudo-client. “From client: Hello World, this is test.” This is the input provided by the user at the client end.

“Listening to heartbeat. . .” is the step wherein the server keeps sending heartbeats to the pseudo-server, to check if there are any requests for itself. “Responded to heartbeat. . .” is the pseudo-server responding to the server with a positive message.

“Response from server: Hello World , this is test.” is the final response from the server back to the client.

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT



```
Command Prompt - python sei
Pseudo-server doesn't have new request...
Pseudo-server doesn't have new request...
Pseudo-server doesn't have new request...
Pseudo-server doesn't have new request...
Server recieved request: Hello World, this is test.
Server preparing response...
Server sent response...
Pseudo-server doesn't have new request...
Pseudo-server doesn't have new request...
```

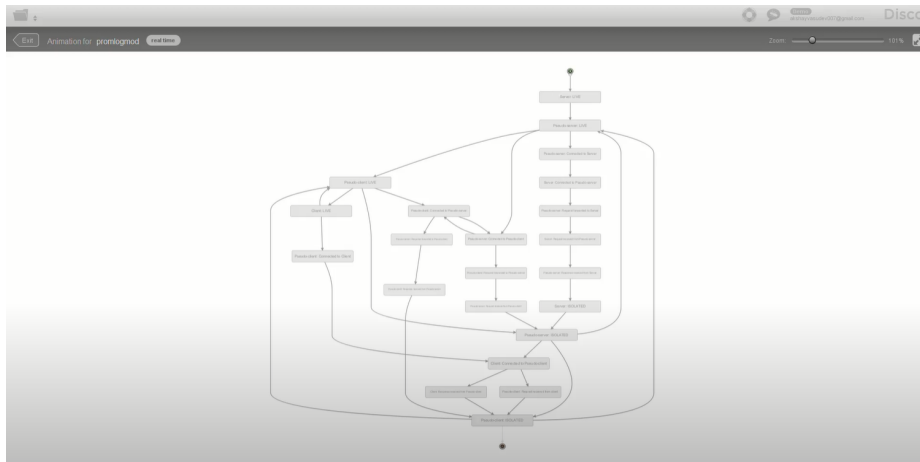
This terminal corresponds to the server wherein the server keeps sending heartbeats to the pseudo-server, as a result, we can see the following message “Pseudo-server doesn’t have new request. . .”. After this step pseudo-server connects to the server when a message is received by it.

The next step is when the message is received, it prepares a response and sends it back to the client, as we can see the following steps are also displayed in the terminal. The next step is for the server to disconnect itself from the pseudo-server and start transmitting heartbeats through the UDP protocol.

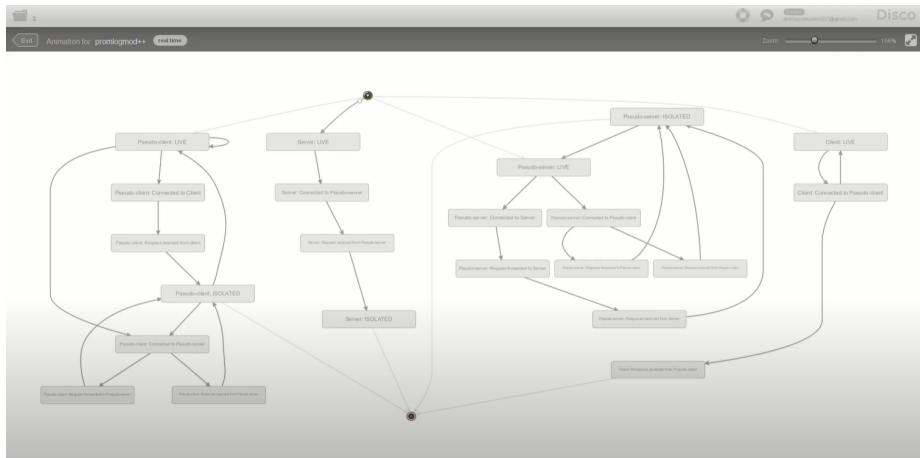


### 2.3.2 Simulation View

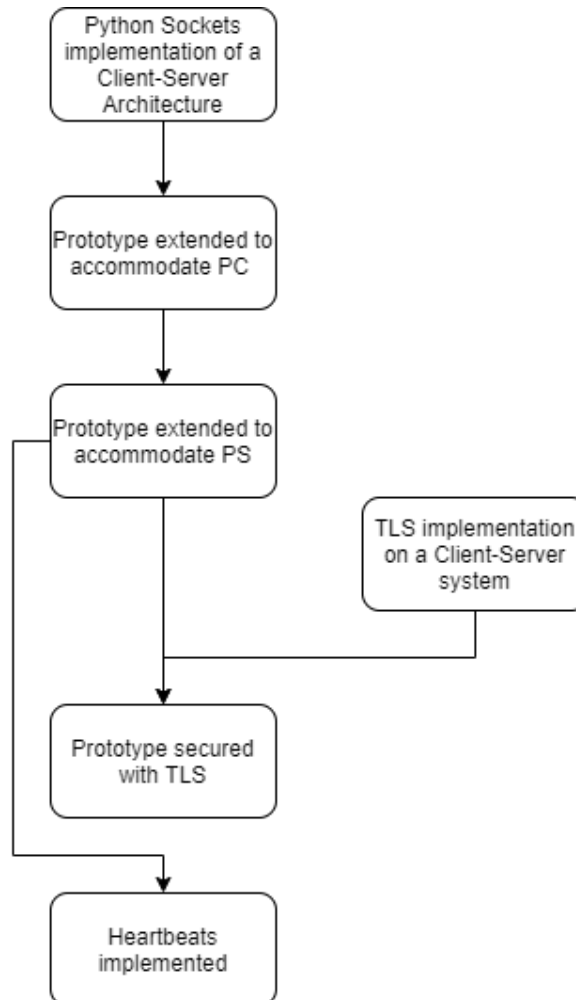
### Process Flow View



## Packet Flow View



## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT



### 2.4 Further Exploration Plans and Timelines

Currently, PSPIT is in a “local system packet simulation” stage, where we programmatically and diagrammatically depict the flow of packets on a packet simulation software, which gives us a clear picture of how the protocol will work, once we move to the multi-system implementation phase, which comes after the current phase.

In the multi-system implementation, four individual systems will be used, instead of one single simulation system, and packets will flow through the use of real-world physical

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

media such as the internet (gateway routers and transmission links) and local area connections (MAC-Address communication).

The systems and their respective pseudo-systems will be connected to a Local Area Network (LAN), through the use of ethernet cables and RJ-45 connectors, to create a secure closed circuit communication medium between the two. Once packets have been sent from the pseudo-machine or vice versa, the local area connection is closed, and hence the main system is never connected to the internet.

The pseudo-system then establishes a connection to the internet and sends the packet to the destination pseudo-system via the internet and the ISP's router network. Similarly, on the receiving end, the pseudo-system receives the packets, shuts itself off from the internet, and connects to the destination system via LAN, in the same way as the sender and the pseudo-sender communicated.

Furthermore, these multi-system communications will be encrypted with the use of TLS, which is Transport Layer Security. TLS is a secure communication encryption standard used worldwide for encrypting internet communications from unintended and/or malicious attackers. TLS uses keys in the form of certificates, cipher and decipher messages, and the certificates are confidentially maintained by each pseudo-system. This TLS security layer acts on top of TCP, with the systems sending their certificates to each other via the protocol, following which messages are encrypted with the destination machine's public key, and decrypted with its private key, as depicted in the diagram.

We will be researching the best configurations of TCP-TLS that can be used as part of our protocol, which was a part of our literature survey as well, to find the most secure and convenient settings package for our use, as well as which configuration works most efficiently on the largest number of systems.

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

The protocol also sees the use of “heartbeats”, which is a term used for when a machine, unprompted, checks for a status change. If no status change is found, then the machine stays in the periodic check state, but if a change is detected, then a series of actions is carried out.

In order to ensure complete host isolation, we could not store the host’s communication details on the pseudo-machine, as it connects to the internet, and if any malicious attacker is able to get access to the pseudo-machine, they could easily start communications with the main system, posing as a pseudo-system.

To prevent this, we made the use of heartbeats in our protocol. Since the communication details are not stored on the pseudo-system, the main system must initiate the communication all on its own. Since the pseudo-system is connected to the internet to check for incoming messages, it cannot be constantly connected to the main system, and hence periodically opens the LAN port to check for any queued messages from the main system, therefore the term “heartbeats”. We will also be working on micro-level security features on the pseudo-system such as partial execution of non-sensitive code to ensure safety, endpoint proxying for additional security. These features will be coming in at the final stage of our protocol, post the paper publication.

The four-system implementation will be our next stage in our protocol’s progression, and will be ready for use in the next review. The heartbeat feature will be available as well, along with the use of TLS certificates.

Post this stage, we will begin working on a ready-to-go IEEE format paper for publication and presentation at a conference. Our primary target is to get our research on the protocol published and presented in a reputed journal and conference respectively.

Finally, we will add all the “caviar” to the protocol, which are the higher-level features for more advanced security as mentioned above. Alongside this, we would also try to

## LOW LEVEL DESIGN AND IMPLEMENTATION DOCUMENT

work with internet regulatory/governance bodies such as IETF, to further fine-tune our protocol, to make it fit for real-world use with sensitive high-value data. We will also work on reducing any additional latencies that arise as a result of using the protocol in place of standard transmission procedures such as bare-bone TCP-TLS.

We would like to take the protocol forward into real-world practical use, rather than just an idea on paper, as we feel that it has vast potential in industries with large volumes of constantly flowing sensitive data, rather than relying on manually controlled off-site data farms, which can help reduce user/company costs.

Post this review, our most immediate task would be preparing the IEEE paper for publication, which we will simultaneously be doing alongside our protocol's progression.

## **Appendix A: Definitions, Acronyms and Abbreviations**

1. TCP: Transmission Control Protocol
2. UDP: User Datagram Protocol
3. IP: Internet Protocol
4. TLS: Transport Layer Security
5. SSL: Secure Socket Layer
6. PSPIT: Pseudo-System Protocol for Information Transfer
7. Client: The node in the network initiating a PSPIT connection.
8. Server: The node in the network responding to the PSPIT client request on the opposite end of the client.
9. Pseudo-systems: The intermediate nodes in the path from the client to the server acting which is practical, router/switches.
10. Pseudo-client: The node which immediately receives the request from the client and forwards the same to the next node, called the pseudo-server. This also relays the response from the pseudo-server to the client.
11. Pseudo-server: The node that receives the request from the Pseudo-client and forwards the same to the server. On receipt of a response from the server, it relays it to the pseudo-client.
12. Heartbeat(s): Repetitive beacons/probes transmitted by the client and the server to check for availability of new data in pseudo-client and pseudo-server respectively.

## **Appendix B: References**

1. Elsadig, Muawia & Fadlalla, Yahia. (2016). Survey on Covert Storage Channel in Computer Network Protocols: Detection and Mitigation Techniques. International Journal of Advances in Computer Networks and Its Security. 6.
2. Pawar, Mohandas & Anuradha, J.. (2015). Network Security and Types of Attacks in Network. Procedia Computer Science. 48. 10.1016/j.procs.2015.04.126.
3. Pandey, Shailja. (2011). MODERN NETWORK SECURITY: ISSUES AND CHALLENGES. International Journal of Engineering Science and Technology. 3.
4. Manu, A. & Rudra, Bhawana & Reuther, Bernd & Vyas, O.. (2011). Design and Implementation Issues of Flexible Network Architecture. 10.1109/CICN.2011.59.
5. Abbasi, Abdul & Muftic, Sead. (2010). CryptoNET: security management protocols. 15-20.
6. Satapathy, Ashutosh & Livingston, Jenila. (2016). A Comprehensive Survey on SSL/TLS and their Vulnerabilities. International Journal of Computer Applications. 153. 31-38. 10.5120/ijca2016912063.
7. Cheng, Pau--chen & Garay, Juan & Herzberg, Amir & Krawczyk, H.. (1998). A security architecture for the Internet Protocol. IBM Systems Journal. 37. 42 - 60. 10.1147/sj.371.0042.
8. Sirohi, Preeti & Agarwal, Amit & Tyagi, Sapna. (2016). A comprehensive study on security attacks on SSL/TLS protocol. 893-898. 10.1109/NGCT.2016.7877537.
9. Chakravarty, Sambuddho & Portokalidis, Georgios & Polychronakis, Michalis & Keromytis, Angelos. (2015). Detection and analysis of eavesdropping in anonymous

- communication networks. International Journal of Information Security. 14. 10.1007/s10207-014-0256-7.
10. James F. Kurose and Keith W. Ross. 2012. Computer Networking: A Top-Down Approach (6th Edition) (6th. ed.). Pearson.
11. Behrouz A. Forouzan and Sophia Chung Fegan. 2002. TCP/IP Protocol Suite (2nd. ed.). McGraw-Hill Higher Education.
12. Atighetchi, Michael & Soule, Nate & Pal, Partha & Loyall, Joseph & Sinclair, Asher & Grant, Robert. (2013). Safe Configuration of TLS Connections - Beyond Default Settings. 6th IEEE Symposium on Security Analytics and Automation (SafeConfig 2013).
13. Castelluccia, Claude & Mykletun, Einar & Tsudik, Gene. (2006). Improving secure server performance by re-balancing SSL/TLS Handshakes. 2006. 26-34. 10.1145/1128817.1128826.
14. Larisch, James & Choffnes, David & Levin, Dave & Maggs, Bruce & Mislove, Alan & Wilson, Christo. (2017). CRLite: A Scalable System for Pushing All TLS Revocations to All Browsers. 539-556. 10.1109/SP.2017.17.
15. Borboruah, Gayatry & Nandi, Gypsy. (2014). A Study on Large Scale Network Simulators. International Journal of Computer Science and Information Technologies. 5. 7318-7322.



**Appendix C: Record of Change History**

#	Date	Document Version No.	Change Description	Reason for Change
1.	09-09-21	1	Initial draft	Initial draft.
2.	12-09-21	2	Heartbeats	Added the heartbeat functionality to maintain the status of availability of systems.
3.	22-09-2021	3	System Check Nodes	Required system failure handling mechanisms.
4.	23-09-2021	4	Dedicated Timers	Additional failure handling