ENSC 427: COMMUNICATION NETWORKS

SPRING 2025

FINAL PROJECT

Evaluating Power and Performance Consumption in a P2P Network While Mitigating a DDOS Attack Within ns-3

https://ariasfernando118.wixsite.com/my-site-1

Kuitenbrouwer, Caleb

Arias, Fernando

Wasilwa, Clarence

301398847,301435230,301429747

cfk2@sfu.ca, fda7@sfu.ca, cww8@sfu.ca

TEAM # 4

List of Figures

List of Tables

Table of Contents

1. ABSTRACT

The project examines how peer-to-peer networks operate when compromised by simulated DDoS attacks and how the network can combat excessive power consumption and performance loss through preventive techniques. Through ns-3, we can apply attack mitigation systems to boost the performance metrics and mitigate the effect of a simulated DDOS attack on the peer-to-peer system. Key components such as network congestion, node processing load, and power efficiency can be modelled and analyzed upon applying the simulation. From these data sets, we can gain insights into how a balance of security and effective energy usage can be derived from a peer-to-peer network while undergoing a simulated DDOS attack.

2. INTRODUCTION

Peer-to-peer (P2P) networks have emerged as a vital component in distributed systems due to their enhanced scalability, resilience, and decentralized control. In P2P networks, clients provide bandwidth, storage space, and computing power resources. As nodes arrive and demand on the system increases, the system's total capacity also increases [1]. Several characteristics make P2P networks robust, and their decentralized structure makes the network efficient against Distributed Denial of Service (DDoS) attacks, which can severely impair network performance and increase power consumption due to the added overhead of malicious traffic. Due to the recent increase in popularity of P2P networks, it is important to analyze the possible ways in which attackers can attack the network, and this project seeks to analyze the impact of DDoS attacks on a simulated P2P network and examine the effectiveness of mitigation techniques in maintaining network performance.

The scope of the project first involves the simulation of a P2P network environment using the ns-3 simulator. It then inspects the network under a DNS amplification DDoS attack, and finally ends with a simulation where various mitigation strategies are performed.

Ultimately, our project aims to contribute to the ongoing discourse on network security and energy efficiency by analyzing the effects a DDoS attack has on the operational efficiency of a P2P system. This is achieved by monitoring network parameters to tally the overall energy consumption during both attack and mitigation phases. This work builds on the foundation laid by previous studies in the field, including a survey that examines recent DDoS mitigation techniques and their comparative analysis [2]. These not only amplify the significance of our research but also guide the development of effective mitigation strategies within our simulation framework.

3. MAIN SECTION(s)

In this work, we employ the ns-3 discrete-event simulator to evaluate peer-to-peer network behaviour under normal operation, under a DNS amplification DDoS attack, and under two mitigation techniques to mitigate the DNS amplification DDoS attack. We configure grid-based mobility for visualization in NetAnim and use FlowMonitor to collect packet-loss statistics.

We compare four scenarios:

1. The baseline P2P network.
2. The same P2P network under a DNS amplification flood.
3. The attack plus a timed rate-limiter at the victim that drops excess packets.
4. The attack plus an IP blacklist relegator at the victim that permanently drops traffic from any source that exceeds the packet-per-second threshold.

This structure allows us to isolate the impact of each condition on throughput, byte-level amplification, packet loss, and implied energy use.

3.1 Peer-to-Peer Network (Scenario 1)

3.1.1 Technical Details and Background Information

A Peer-to-peer (P2P) network is a decentralized network architecture in which participants, called peers, interact directly with each other without needing a central authority or server [3]. In a P2P network, each participant acts as a client and a server, enabling them to share resources and services directly with other peers. This decentralization eliminates single points of failure and allows for more resilient and scalable networks [4]. It also brings about efficient resource utilization and brings some cost savings due to the bypass of expensive dedicated servers and a centralized infrastructure. Shown in the next page is a sample P2P Network Topology showing the interaction of the various peers.

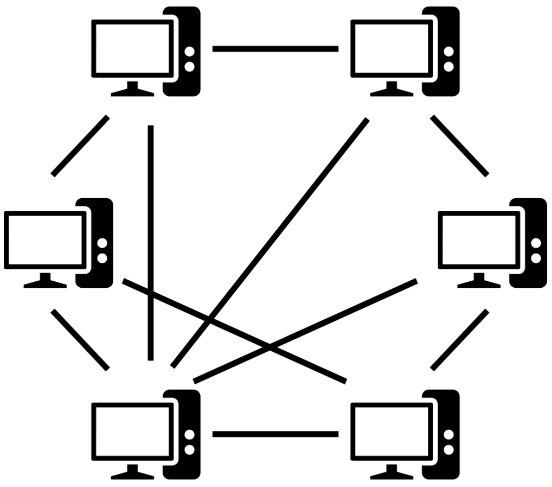
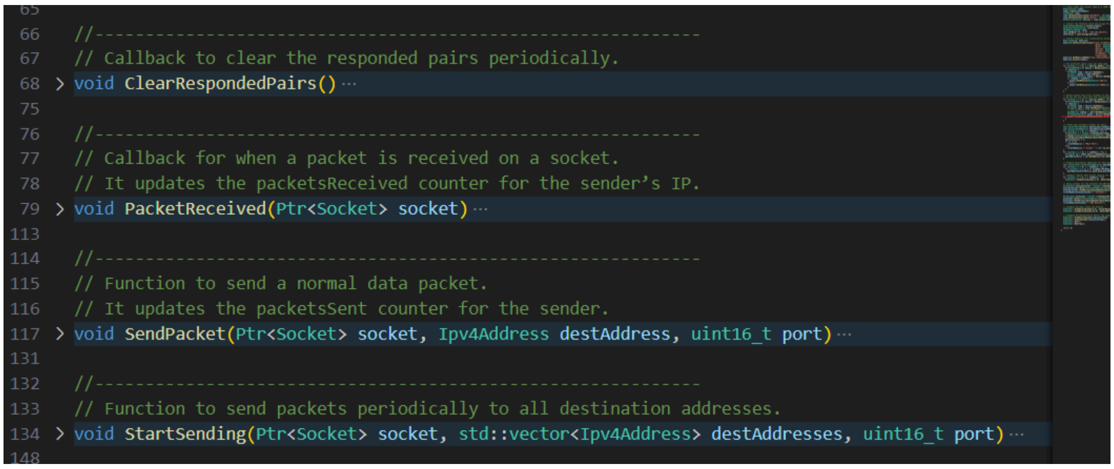


Figure 1: Peer-to-peer Network Topology [5]

P2P Simulation Setup

The Peer-to-Peer Network simulation comprises multiple components that contribute to creating the network's behaviour.

Helper functions were created to simulate the behavior of peers when sending and receiving packets, as well as to perform the action of sending packets from one peer to another. Then, the helper functions were used to create the P2P network simulation



```
//------------------------------------------------------------
// Callback to clear the responded pairs periodically.
void ClearRespondedPairs() ···

//------------------------------------------------------------
// Callback for when a packet is received on a socket.
// It updates the packetsReceived counter for the sender's IP.
void PacketReceived(Ptr<Socket> socket) ···

//------------------------------------------------------------
// Function to send a normal data packet.
// It updates the packetsSent counter for the sender.
void SendPacket(Ptr<Socket> socket, Ipv4Address destAddress, uint16_t port) ···

//------------------------------------------------------------
// Function to send packets periodically to all destination addresses.
void StartSending(Ptr<Socket> socket, std::vector<Ipv4Address> destAddresses, uint16_t port) ···
```

Figure 2: P2P Network Simulation Helper Functions

In Figure 2, we identified three helper functions: “PacketReceived”, “SendPacket”, and “StartSending”. We use these functions in the main function to create the P2P network.

The **SendPacke**t function grabs as input a pointer to the sender peer socket, the receiver peer’s address, and the receiver’s port. It creates a 1024-byte packet, which is then sent from the sender socket to the specified address and port via the UDP protocol. Finally, it retrieves the sender’s information and displays it in the terminal.

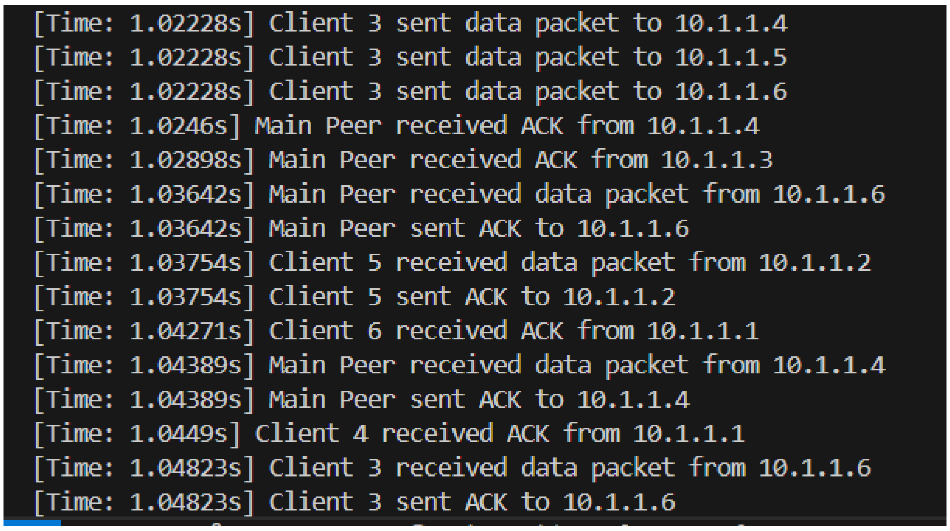
The **PacketReceived** function grabs a pointer to the receiver peer socket as input. It retrieves the incoming packet and the sender's address from that socket. Then, based on the packet size, it determines whether the received message is an acknowledgement or a data packet from a peer. The function logs details about the peer, sender address, and packet type, and finally sends an ACK back to the sender if the packet is data.

The **StartSending** function takes as input a pointer to the sender's socket, a list of destination peer addresses, and the receiver's port. It then checks whether the sending peer is the "Main Peer" or an ordinary peer. In our P2P model, the Main Peer is the node that handles larger volumes of traffic. This function establishes the system's data flow by scheduling packet sends to all destinations at randomized delays, and provides the Main Peer with a shorter delay window to reflect its higher-traffic role.

Finally, the **ClearRespondedPairs** function clears the global variables used to track sender and receiver address pairs. The main function creates multiple rounds of simulation, and we need to use ClearRespondedPairs at the beginning of each round. These repeated rounds help us collect data and draw better conclusions about network performance

3.1.2 Simulation Results and Analysis

Upon running the basic peer-to-peer simulation, we can generate associated logs to analyze the behaviour of the peers and the traffic involved in the system.

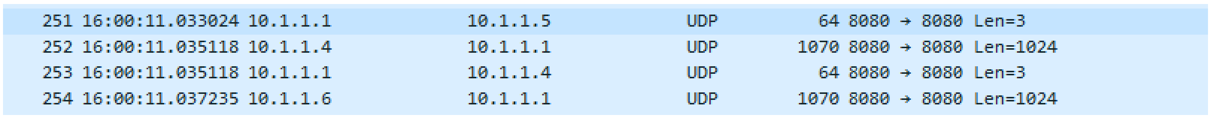


```
[Time: 1.02228s] Client 3 sent data packet to 10.1.1.4
[Time: 1.02228s] Client 3 sent data packet to 10.1.1.5
[Time: 1.02228s] Client 3 sent data packet to 10.1.1.6
[Time: 1.0246s] Main Peer received ACK from 10.1.1.4
[Time: 1.02898s] Main Peer received ACK from 10.1.1.3
[Time: 1.03642s] Main Peer received data packet from 10.1.1.6
[Time: 1.03642s] Main Peer sent ACK to 10.1.1.6
[Time: 1.03754s] Client 5 received data packet from 10.1.1.2
[Time: 1.03754s] Client 5 sent ACK to 10.1.1.2
[Time: 1.04271s] Client 6 received ACK from 10.1.1.1
[Time: 1.04389s] Main Peer received data packet from 10.1.1.4
[Time: 1.04389s] Main Peer sent ACK to 10.1.1.4
[Time: 1.0449s] Client 4 received ACK from 10.1.1.1
[Time: 1.04823s] Client 3 received data packet from 10.1.1.6
[Time: 1.04823s] Client 3 sent ACK to 10.1.1.6
```

Figure 3: Peer-to-peer Network Logs

From Figure 3, we can observe the sequential behaviour of the clients as they send packets to one another in a random pattern. These packets sent out are then responded to by the peers through smaller

‘ACK’ packets. The ACK messages ensure that both interacting peers have a guarantee that their exchange was successful. These sequential interactions are scheduled every five seconds and last typically for 2-3 seconds. By using Wireshark, we can take the analysis further by capturing the packet exchange of these nodes.



```
251 16:00:11.033024 10.1.1.1     10.1.1.5     UDP       64 8080 → 8080 Len=3
252 16:00:11.035118 10.1.1.4     10.1.1.1     UDP     1070 8080 → 8080 Len=1024
253 16:00:11.035118 10.1.1.1     10.1.1.4     UDP       64 8080 → 8080 Len=3
254 16:00:11.037235 10.1.1.6     10.1.1.1     UDP     1070 8080 → 8080 Len=1024
```

Figure 4: Peer-to-peer Network Packet Capture

From the above packet capture, we can see clients exchanging packets with payloads the size of 1024 bytes each, with a header length of 46 bytes. Then the associated ACK sent back contains a much smaller payload of only 3 bytes and a header size of 10 bytes. Typical behaviour of a peer-to-peer network was achieved through both the logs and packet capture. To get a visual aid on the network, a netanim of the network diagram was generated.

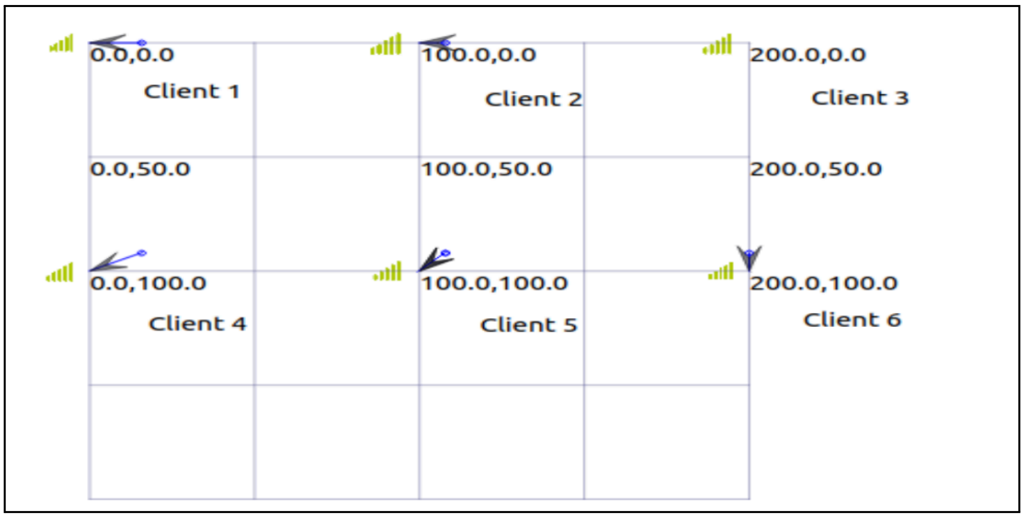


Figure 5: Client 3 Interaction Peer-to-peer Network

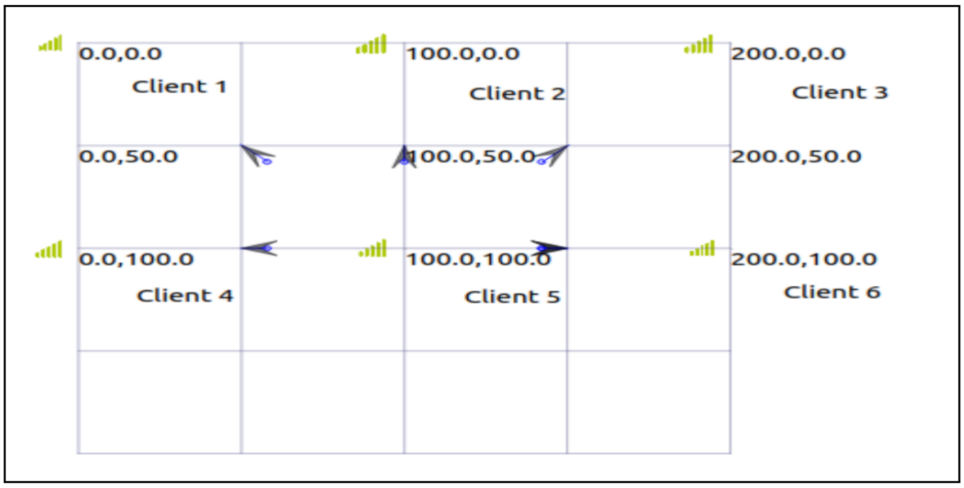


Figure 6: Client 5 Interaction Peer-to-peer Network

Mysteriously, unlike the generated logs and packet capture, the interaction of the clients is seen to be broadcast but only accepted at the target address. This visual shows the basic interaction of the clients, as every few milliseconds, a random client acknowledges an exchange it had or sends a packet to a new client. We believe this broadcasting display to be a visual glitch with the animation software, as the byte exchange is accurately calculated using Flowmotion.

Node	**Bytes Sent**	**Bytes Received**	**Packets Sent**	**Packets Received**	**Power Consumption (Given as Energy in mJ)**
Main Peer	12,996	12,996	12	12	181.944
Client 2-6	64,980	64,980	60	60	909.72
				Total	1,091.664

Table 1: Scenario 1 Results

This byte-sized data was collected using the ns3 Flowmotion tool implemented within the code that tracks the peer's behaviour and scans packet values as they are transmitted. To derive the number of packets exchanged, global counters were used within the code that iteratively increment based on the IP addresses of the sending and receiving clients. From the table above, we can see the normal interaction of the peers as each one sends out 12 Packets (1070 bytes per) to one another during the designated periods. From these byte values, we derived a power consumption value equivalent through a conversion process. It should be noted that the packet sent and received columns only reflect the number of non-ACK new packets generated, even though the bytes columns involved include those 13-byte packets.

3.2 DDoS Network (Scenario 2)

3.2.1 Technical Details and Background Information

A Distributed Denial of Service (DDoS) attack can be described as a directed attack launched indirectly through many compromised computing systems on the availability of target services with the aim of the attack not being to crack system authentication or to achieve unauthorized system access but rather, being to degrade the performance of a network and to make it impossible for it to provide requested service to its legitimate clients [6]. DDoS attacks are among the leading cyber attacks that take place daily and thus further highlight the importance of their choice in the simulation [7]. Below shows a visual of other common cyber attacks.

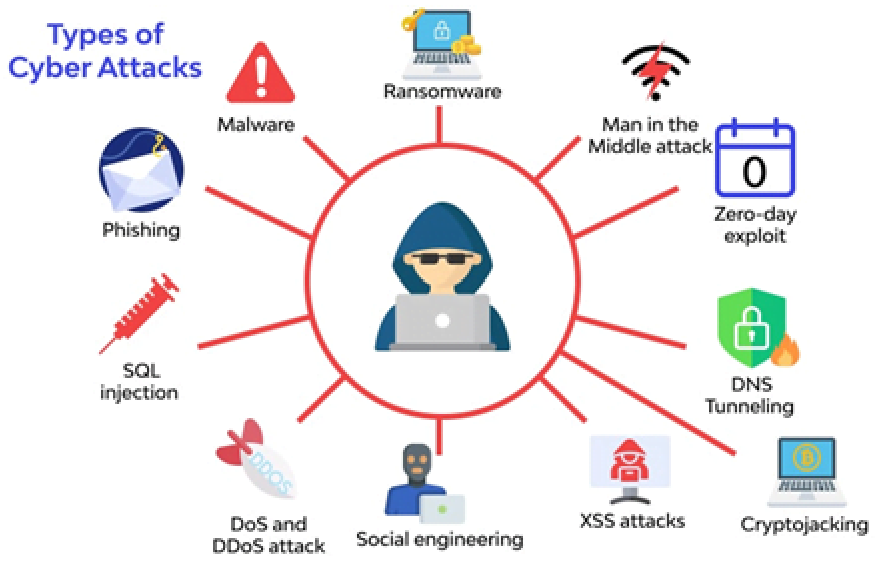


Figure 7: Common Cyber Attacks [8]

DDoS attacks can be broadly classified into three categories [9]:

1. Volumetric (packet or byte floods, e.g., UDP, ICMP)
2. Protocol (exploits server or network equipment state, e.g., SYN flood)
3. Application-layer (target-specific services, e.g., HTTP GET floods)

For this work, the simulation focused on a DNS amplification attack. A DNS resolver translates domain names into IP addresses, allowing users to access websites. In a DNS amplification attack, an adversary sends small spoofed DNS queries to open resolvers, forging the victim's IP as the source. The resolver then replies with a much larger response, in a larger magnitude to the original query size, thus flooding the victim's link and resources. DNS amplification typically ranges from 20 to 100 times the original query's size. The figure below outlines a visual of a DNS amplification attack.

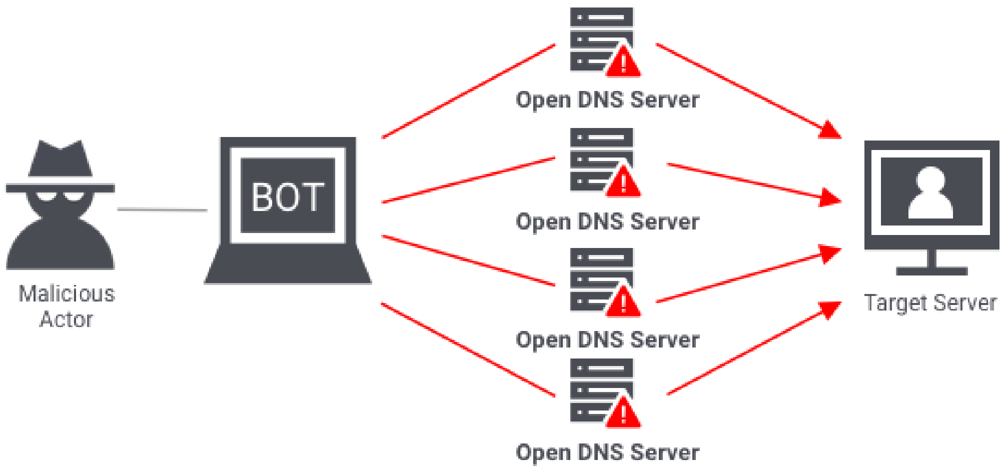
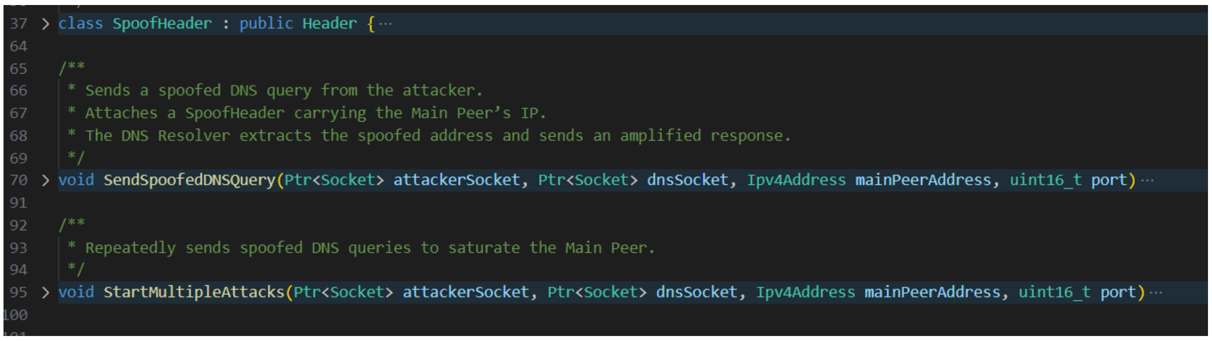


Figure 8: DNS Amplification DDoS Attack [10]

For the simulation setup, building on the P2P baseline, we add two nodes: an Attacker and a DNS Resolver. The attacker emits 60-byte spoofed queries (with a 4-byte custom header) every 6.5 ms starting at t = 10 s; the resolver replies with 8 000-byte responses to the spoofed ("victim") IP. Both use UDP sockets on port 8080, with CSMA channel parameters unchanged. The victim (Main Peer) continues normal traffic and ACK handling. FlowMonitor and NetAnim remain enabled,

DNS Amplification DDoS Simulation Setup

To create the DNS Amplification DDoS, we created a class to implement the required spoofing. We then use helper functions within the main function to implement the attack.



```
> class SpoofHeader : public Header {···

  /**
   * Sends a spoofed DNS query from the attacker.
   * Attaches a SpoofHeader carrying the Main Peer's IP.
   * The DNS Resolver extracts the spoofed address and sends an amplified response.
   */
> void SendSpoofedDNSQuery(Ptr<Socket> attackerSocket, Ptr<Socket> dnsSocket, Ipv4Address mainPeerAddress, uint16_t port)···

  /**
   * Repeatedly sends spoofed DNS queries to saturate the Main Peer.
   */
> void StartMultipleAttacks(Ptr<Socket> attackerSocket, Ptr<Socket> dnsSocket, Ipv4Address mainPeerAddress, uint16_t port)···
```

Figure 9: Code to implement DDoS

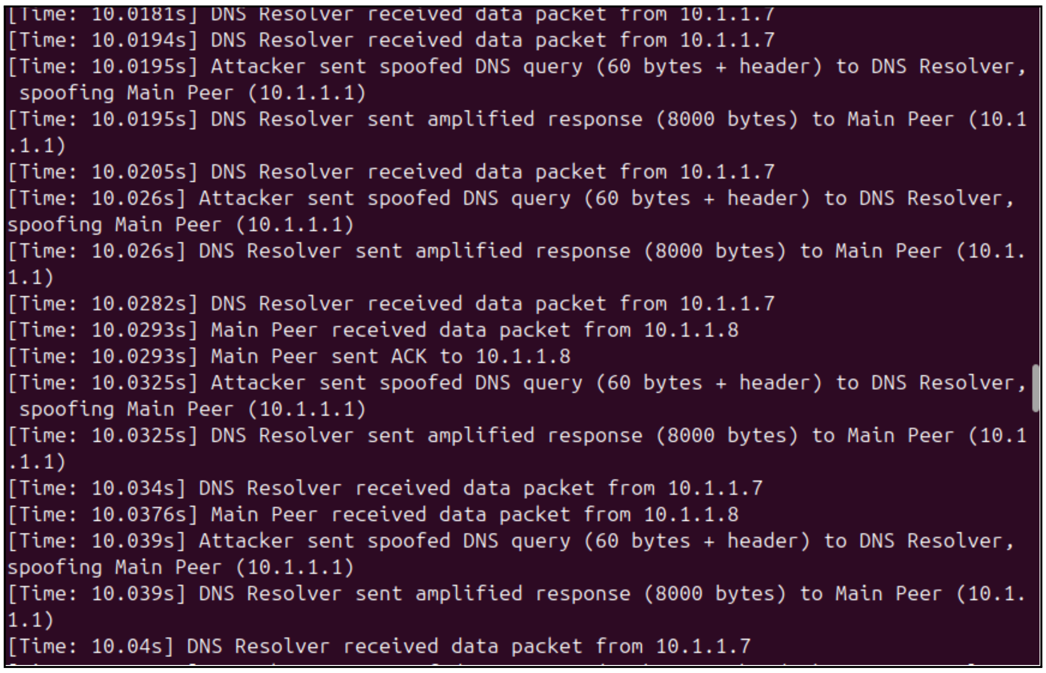
Figure 9 shows the code used to create the DDoS attack. We have two helper functions to implement the attack, as well as a class that inherits from the Header class to perform spoofing.

The **SpoofHeader** class inherits from Header, so we can use it to modify the packet's source address when the attacker sends it. We use this class to simulate the spoofing component of a DNS amplification DDoS attack.

The **SendSpoofQuery** function takes as input two socket pointers, for the attacker node and the DNS resolver, along with the victim's IP address and port. Recall that in this paper, we define the Main Peer as the peer handling the most network traffic. **SendSpoofQuery** uses the "SpoofHeader" to replace the attacker's source address with the Main Peer's address, then sends a DNS request that elicits an amplified response to the victim about 80 times larger than the original query.

Finally, the **StartMultipleAttacks** function takes the same inputs as **SendSpoofQuery** and repeatedly invokes it at a defined time interval to launch successive attacks.

3.2.2 Simulation Results and Analysis



```
[Time: 10.0181s] DNS Resolver received data packet from 10.1.1.7
[Time: 10.0194s] DNS Resolver received data packet from 10.1.1.7
[Time: 10.0195s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
 spoofing Main Peer (10.1.1.1)
[Time: 10.0195s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1
.1.1)
[Time: 10.0205s] DNS Resolver received data packet from 10.1.1.7
[Time: 10.026s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
spoofing Main Peer (10.1.1.1)
[Time: 10.026s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1.
1.1)
[Time: 10.0282s] DNS Resolver received data packet from 10.1.1.7
[Time: 10.0293s] Main Peer received data packet from 10.1.1.8
[Time: 10.0293s] Main Peer sent ACK to 10.1.1.8
[Time: 10.0325s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
 spoofing Main Peer (10.1.1.1)
[Time: 10.0325s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1
.1.1)
[Time: 10.034s] DNS Resolver received data packet from 10.1.1.7
[Time: 10.0376s] Main Peer received data packet from 10.1.1.8
[Time: 10.039s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
spoofing Main Peer (10.1.1.1)
[Time: 10.039s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1.
1.1)
[Time: 10.04s] DNS Resolver received data packet from 10.1.1.7
```

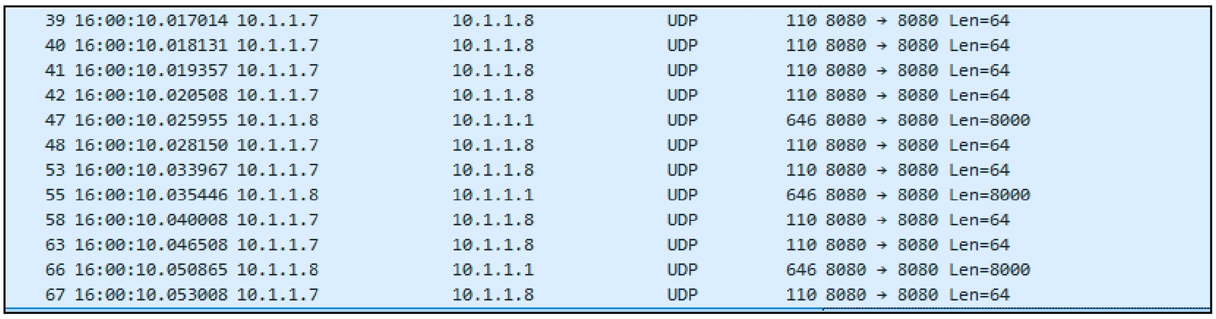
Figure 10: DDoS Attack Logs

Figure 10 presents a snip of the logs of our simulation. This figure shows how the network behaves under a DDoS attack. We can appreciate how the attacker 10.1.1.7 sends a spoofed query with the victim's IP address to the DNS resolver, and how this server answers back to the main peer (the victim) as the server thought the main peer was the one sending the query. In the logs presented, we also notice how the main peer is receiving the packets from the DNS resolver and how the attacker node is sending multiple attacks.

Node	**Bytes Sent**	**Bytes Received**	**Packets Sent**	**Packets Received**	**Power Consumption (Given as Energy in mJ)**
Attacker	42,504	62	462	0	340.404
DNS Resolver	3,708,998	42,566	462	462	29,927.38

Table 2: Scenario 2 Results

In Table 2 we can appreciate how the DNS amplification attack was performed. The attacker sent 42,504 bytes to the DNS Resolver, which then answered and sent 3,708,998 bytes to the main peer, giving us an amplification of 87.26. We can also see in Table 2 that the attacker didn't receive a significant amount of bytes, which confirms that the attack didn't reflect back to the attacker and was successfully directed to the victim peer. In general, we can say that the attack simulation was successful. Finally, we can appreciate that the energy consumption of the attacker was significantly lower than the energy used by the DNS server. This shows us how attackers can perform big attacks without saturating their system.



```
39 16:00:10.017014 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
40 16:00:10.018131 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
41 16:00:10.019357 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
42 16:00:10.020508 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
47 16:00:10.025955 10.1.1.8    10.1.1.1    UDP    646 8080 → 8080 Len=8000
48 16:00:10.028150 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
53 16:00:10.033967 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
55 16:00:10.035446 10.1.1.8    10.1.1.1    UDP    646 8080 → 8080 Len=8000
58 16:00:10.040008 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
63 16:00:10.046508 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
66 16:00:10.050865 10.1.1.8    10.1.1.1    UDP    646 8080 → 8080 Len=8000
67 16:00:10.053008 10.1.1.7    10.1.1.8    UDP    110 8080 → 8080 Len=64
```

Figure 11: DDoS Attack Packet Capture

Figure 11 comes from a pcap file generated from the DDoS attack simulation code. The pcap was opened on Wireshark to get the packet traces shown in the figure. We can see that the DDoS attack was behaving as expected: attacker node 10.1.1.7 sends a UDP packet with a payload of size 64 to the DNS resolver 10.1.1.8, which then sends a response with a payload of size 8000 to the main peer 10.1.1.1 (victim). It's important to highlight that even though the figure shows that the sender is the node 10.1.1.7, the node is internally performing spoofing. Even if it's not shown in the picture, we know that the spoofing was successfully performed, as the DNS server responded to the victim and not to the attacker.

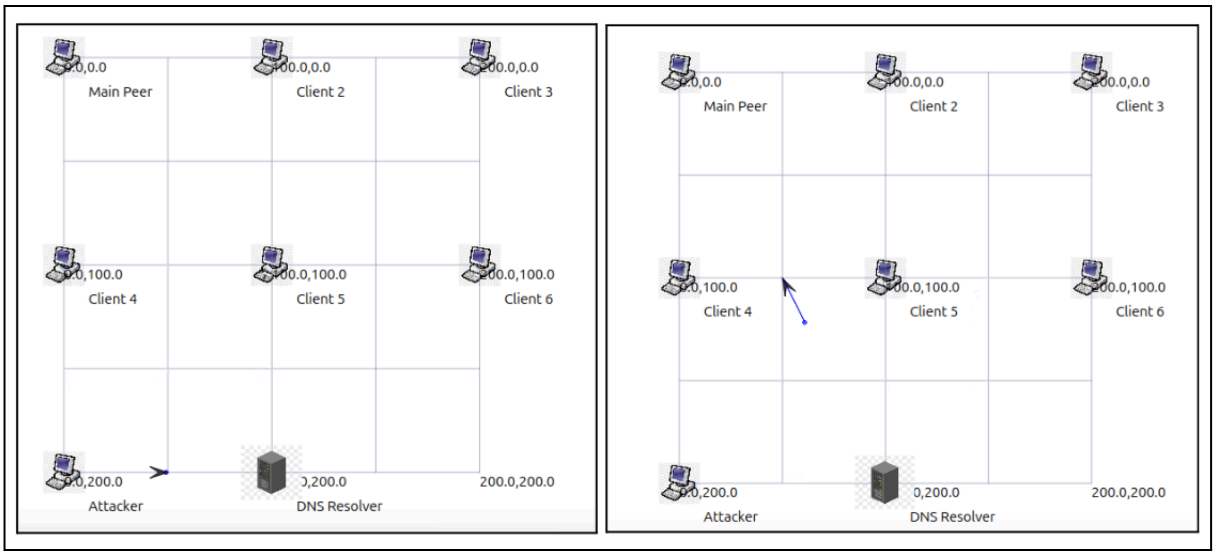


Figure 12: DDoS Attack Netanim Animation

Figure 12 shows a visual representation of the DNS Amplification DDoS attack. We can notice how first the attacker sends the attack to the DNS Resolver, then the DNS Resolver sends the response to the Main Peer. This clearly shows how the attacker successfully performed spoofing to carry out the attack.

Scenario 1 Main Peer Packets Received	Scenario 2 Main Peer Packets Received
12	469

Scenario 2 Packets Lost	Scenario 2 Bytes Lost
5	40,140

Table 3: Main peer Results under Scenario 1 & 2

Table 3 shows us the results of targeting the DDoS attack to the main peer. In the first scenario, where the main peer was only under a normal peer-to-peer network, the main peer only received 12 packets. In the second scenario, it received 469. It's important to mention that even though the table does not show the size of the packets, the packets received in the first scenario had a size of 1024, while in the second scenario, the packet size was 8000. This increases the computational power required by the main peer to process all the received packets. As a result of the DDoS attack, the main peer lost 5 packets, which represent 40,140 bytes of data.

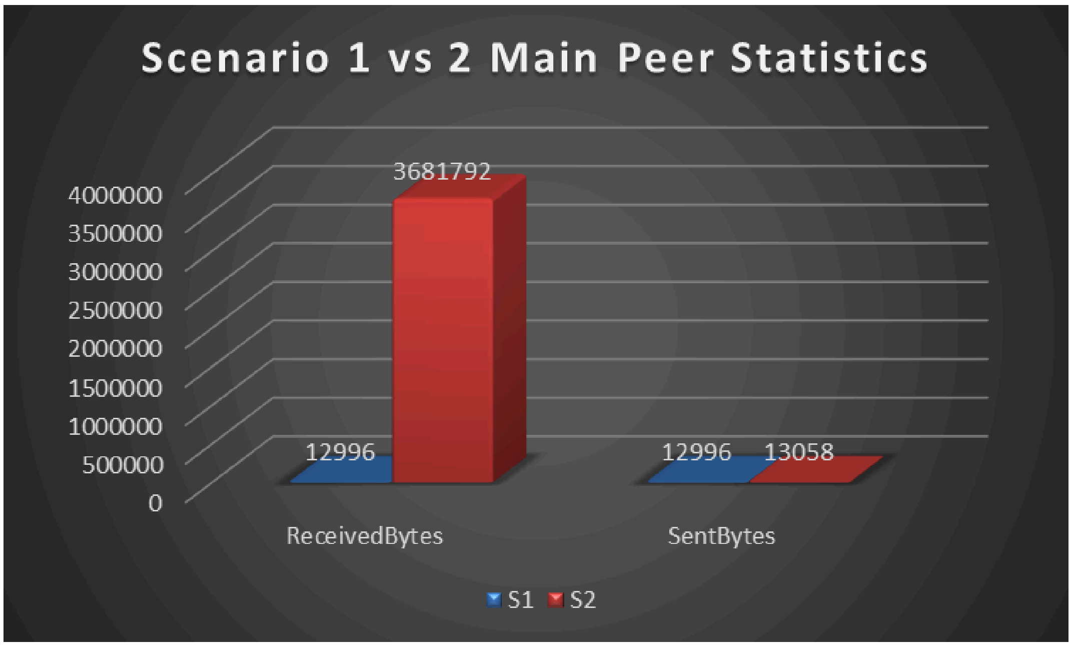


Figure 13: Main Peer Statistics under Scenario 1 & 2

Figure 13 shows us the bytes received and sent by the main peer under a normal peer-to-peer network and a peer-to-peer network during a DNS amplification attack targeting the main peer, as the main peer is the one that manages the most traffic of the network. We can appreciate that the DDoS attack was successfully simulated, as the main peer receives a large amount of bytes in the second scenario. Something important to notice is that even though the main peer is under a DDoS attack in the second scenario, the bytes sent by it increase instead of decreasing. The increase in bytes could be due to ACKs sent to the DNS resolver. The expected result was for the main peer to decrease the amount of bytes sent; this unexpected result suggests that the simulation needs some adjustments.

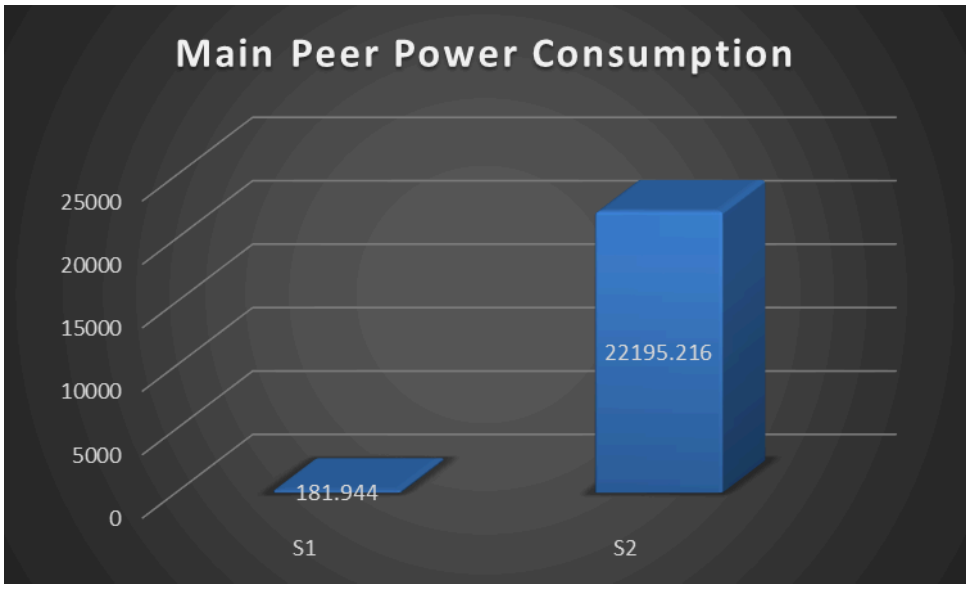


Figure 14: Main Peer Power Consumption

Figure 14 shows the power consumption registered under Scenario 1, the normal peer-to-peer network, and Scenario 2, peer-to-peer under a DDoS attack. We can clearly see that the energy consumption significantly increases when the main peer is under attack. This is due to the large amount of data being received and processed by the main peer. These large amounts of energy consumed for processing big chunks of data could make the main peer collapse and stop working.

3.3 Mitigation Strategies (Scenarios 3 and 4)

3.3.1 Technical Details and Background Information

From the data presented in the previous section, through the utilization of mitigation techniques, we aim to reduce the effect of the malicious overload on the main peer, ensuring a boost to the performance and power consumption of the peer-to-peer network. DDOS mitigation ranges in forms that all work to process, respond and reduce the effect of Distributed Denial of Service attacks on a network [11]. Unlike other precautionary security measures, such as firewalls or encryption, mitigation techniques are reactionary in that they work to minimize an already prevalent attack. Mitigation looks to restore the service performance to the victim user by singling out the source of the attacker's traffic and reducing or eliminating its effect.

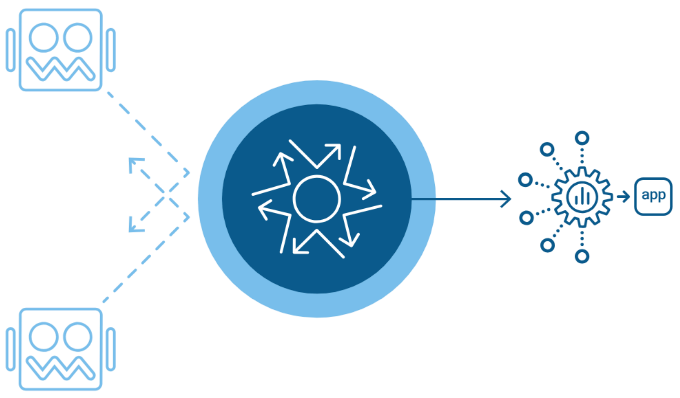
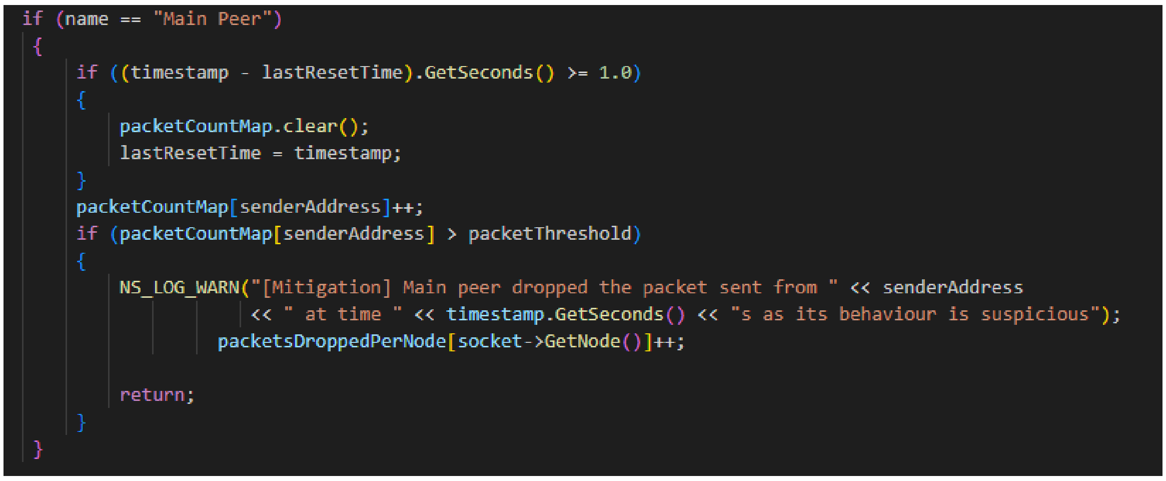


Figure 15: Mitigation Visual App Protection [12]

In the context of the project, we wish to integrate these techniques into our peer-to-peer system network to protect the integrity of the main peer service. In the client-server model, these mitigation techniques are implemented at the network edge to filter generated traffic in its early transmission. Due to the decentralized nature and the inherent lack of a central authority in the network, we created proves challenging to provide an all-encompassing mitigation system [11]. As a result, focus was instead laid on implementing an embedded sub-system directly onto an individual predetermined targeted node to examine its performance.

Mitigation Simulations Setup

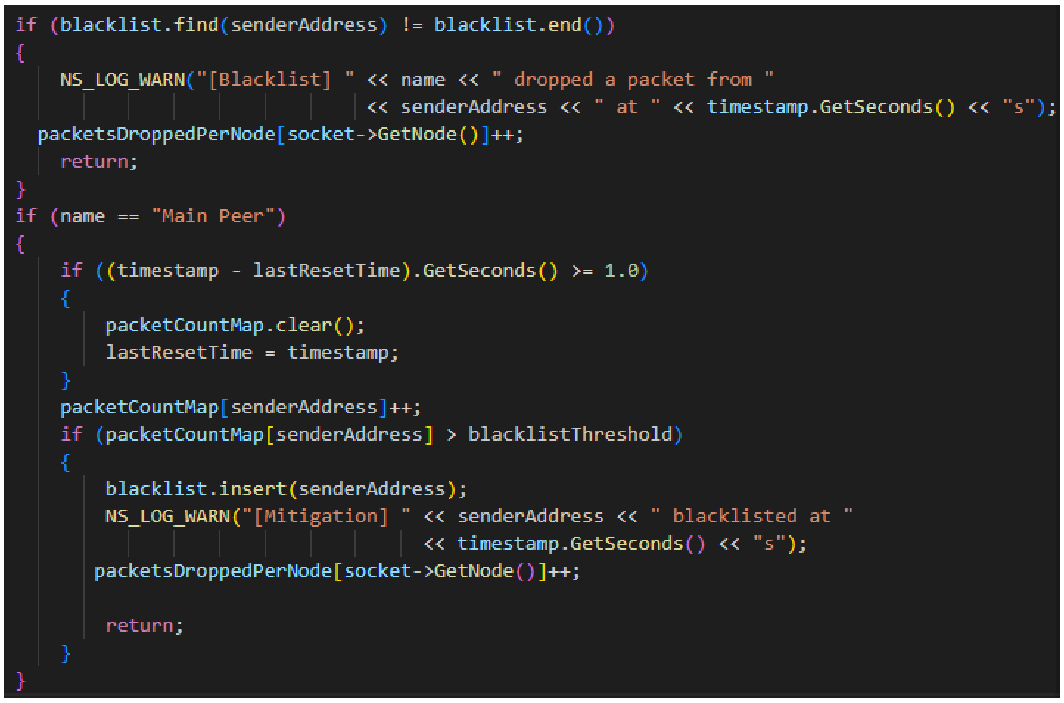
A comparison between two established methods of mitigation was devised to examine each one's performance in regards to reduction in both power consumption and malicious packet reception rate. This was a two-tier approach to mitigation in that both functions built on and used similar elements to both filter and drop malicious packets. Below we can see both mitigation strategies embedded into to the receiver function of the code.



```
if (name == "Main Peer")
 {
     if ((timestamp - lastResetTime).GetSeconds() >= 1.0)
     {
         packetCountMap.clear();
         lastResetTime = timestamp;
     }
     packetCountMap[senderAddress]++;
     if (packetCountMap[senderAddress] > packetThreshold)
     {
         NS_LOG_WARN("[Mitigation] Main peer dropped the packet sent from " << senderAddress
                        << " at time " << timestamp.GetSeconds() << "s as its behaviour is suspicious");
                  packetsDroppedPerNode[socket->GetNode()]++;

         return;
     }
 }
```

Figure 16: Rate Limit Mitigation



```
if (blacklist.find(senderAddress) != blacklist.end())
{
    NS_LOG_WARN("[Blacklist] " << name << " dropped a packet from "
                             << senderAddress << " at " << timestamp.GetSeconds() << "s");
  packetsDroppedPerNode[socket->GetNode()]++;
    return;
}
if (name == "Main Peer")
{
    if ((timestamp - lastResetTime).GetSeconds() >= 1.0)
    {
        packetCountMap.clear();
        lastResetTime = timestamp;
    }
    packetCountMap[senderAddress]++;
    if (packetCountMap[senderAddress] > blacklistThreshold)
    {
        blacklist.insert(senderAddress);
        NS_LOG_WARN("[Mitigation] " << senderAddress << " blacklisted at "
                                  << timestamp.GetSeconds() << "s");
       packetsDroppedPerNode[socket->GetNode()]++;

        return;
    }
}
```

Figure 17: BlackList Mitigation

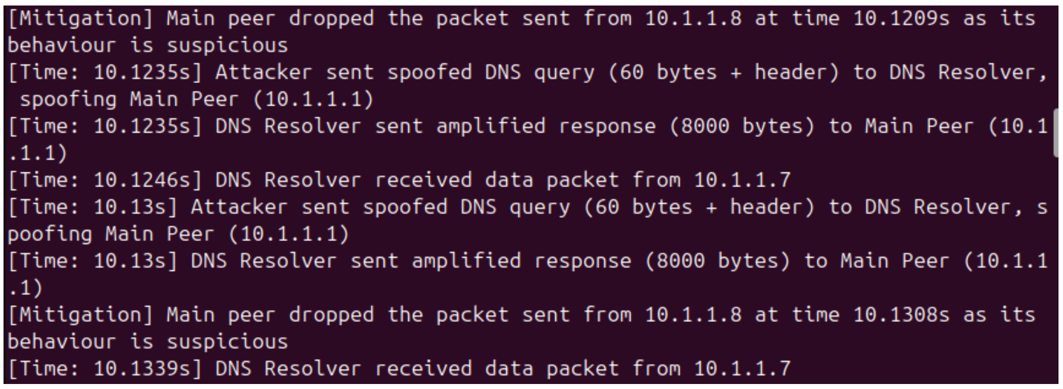
From Figure 16, we can see the implementation of the **Rate-Limiting Mitigation** technique specifically bound to that of the main peer. Throughout its peer-to-peer communication, this node maintains a count of the number of packets received from each sender within a 1-second window. If any peer were to exceed a configurable packet threshold, in this case 10 packets/second (10700bps), the packet is immediately dropped, and the event is then logged. At its core, this mitigation strategy serves as a real-time flood detector that ensures short bursts of unusually high traffic are dropped. This protects the integrity of the main peers' processing capacity while mitigating power consumption.

From Figure 17, the use of a **Blackllist Mitigaiton** strategy that is embedded into the main peers receiving module. If the packet count for a particular sender is equivalent or exceeds a higher blacklist threshold the sender is permanently added to a blacklist by the victim node. Any further packets from this sender are dropped without the need for follow-up inspection. This second tier of mitigation allows the node to escalate its response from a simple temporary rate-limiting to long-term blocking. This leads to a reduction in performance and power cost for repeated attacks from a single attacker.

By analyzing only the local traffic behaviour of this victim node, we can note that this method is effective for the decentralized communication setup within this peer-to-peer network.

3.3.2 Simulation Results and Analysis

In scenario three, we implemented the rate-limiting mitigation as an initial strategy to combat the flooding done to the Main Peer. As before, we can capture the behaviour of this technique using logs embedded into this security system.

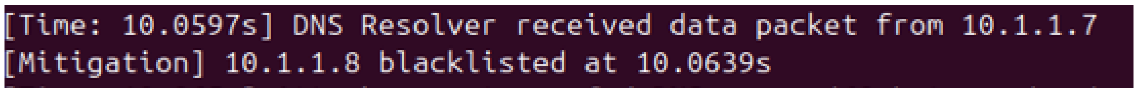


```
[Mitigation] Main peer dropped the packet sent from 10.1.1.8 at time 10.1209s as its
behaviour is suspicious
[Time: 10.1235s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
 spoofing Main Peer (10.1.1.1)
[Time: 10.1235s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1
.1.1)
[Time: 10.1246s] DNS Resolver received data packet from 10.1.1.7
[Time: 10.13s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver, s
poofing Main Peer (10.1.1.1)
[Time: 10.13s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1.1
.1)
[Mitigation] Main peer dropped the packet sent from 10.1.1.8 at time 10.1308s as its
behaviour is suspicious
[Time: 10.1339s] DNS Resolver received data packet from 10.1.1.7
```

Figure 18: Rate-Limit Mitigation Output

During the instance of attack, we can see that the mitigation strategy drops packets from the suspicious IP address that had exceeded the sending limit threshold. As mentioned in the section prior, this works on a case-by-case basis, so each packet has to be processed and identified to exceed the limit before being received by the Main peer. This, in turn, allows for some malicious packets to then get through.

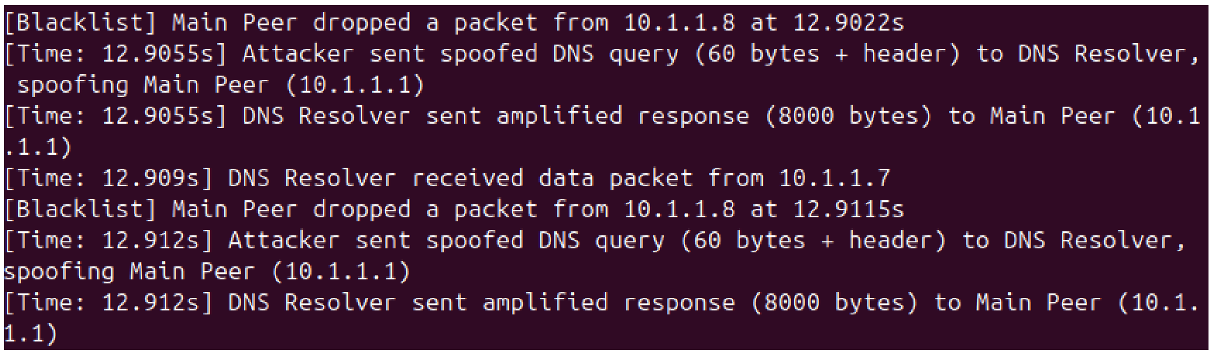
In scenario four, we wish to increase the security by raising the mitigation strategy up a tier by implementing a permaban system on malicious IP addresses. The Blacklist mitigation strategy logs can be seen below.



```
[Time: 10.0597s] DNS Resolver received data packet from 10.1.1.7
[Mitigation] 10.1.1.8 blacklisted at 10.0639s
```

Figure 19: Initial Blacklist Assertion

As expected we can see the initial recognition from the main peer that the same IP address is causing a malicious flood and denotes it onto its registered 'blacklist'.



```
[Blacklist] Main Peer dropped a packet from 10.1.1.8 at 12.9022s
[Time: 12.9055s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
 spoofing Main Peer (10.1.1.1)
[Time: 12.9055s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1
.1.1)
[Time: 12.909s] DNS Resolver received data packet from 10.1.1.7
[Blacklist] Main Peer dropped a packet from 10.1.1.8 at 12.9115s
[Time: 12.912s] Attacker sent spoofed DNS query (60 bytes + header) to DNS Resolver,
spoofing Main Peer (10.1.1.1)
[Time: 12.912s] DNS Resolver sent amplified response (8000 bytes) to Main Peer (10.1.
1.1)
```

Figure 20: Blacklist Mitigation Output

After asserting this IP on the blacklist any further packets received are dropped and this can be seen in the logs designated above. From Flowmotion, we can compare the performance of both mitigation strategies.

	Main Peer Packets Dropped	**Total Main Peer Packets Received**
Scenario 3	441	27
Scenario 4	454	15

Table 4: Scenario 3 and 4 results

If we factor out the packets collected from the normal peer-to-peer communication, tier 1 mitigation has a successful filtration rate upon detection of 96.5%, whilst the tier 2 implementation has a rate of 99.3%. The elimination of having the mitigation system work on a case-by-case basis results in an overall improvement to the detection method of malicious packets.

	Packets Lost	**Bytes Lost**
Scenario 3	5	40,140
Scenario 4	6	48,168

Table 5: Scenario 3 and 4 results

With the additional time that's needed to process the packets to identify a burst, coming from the same IP, packet loss is induced. As we can this is then reduced by implementing the tier 2 mitigation, reducing load on the processing requirements of the main peer.

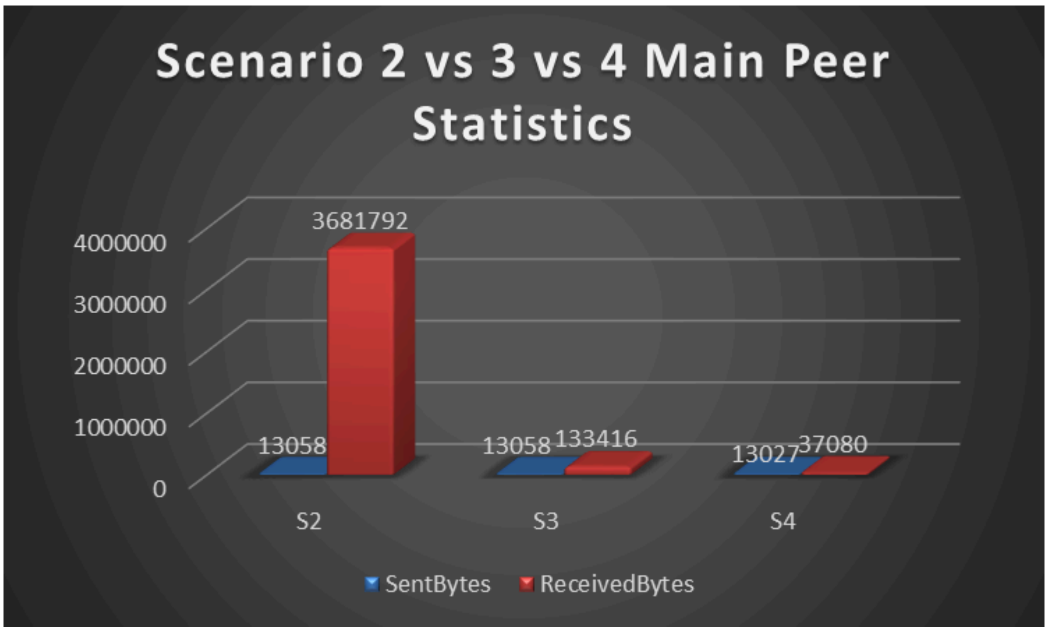


Figure 21: Three Scenarios Main Peer Byte Comparison

From the graph above, we can note that as the mitigation system becomes more advanced, the total load that the main peer receives gradually decreases. This is due to the filtration of the packets coming from the DNS resolver, the immediate inclination of the system to drop the packets before it becomes processed.

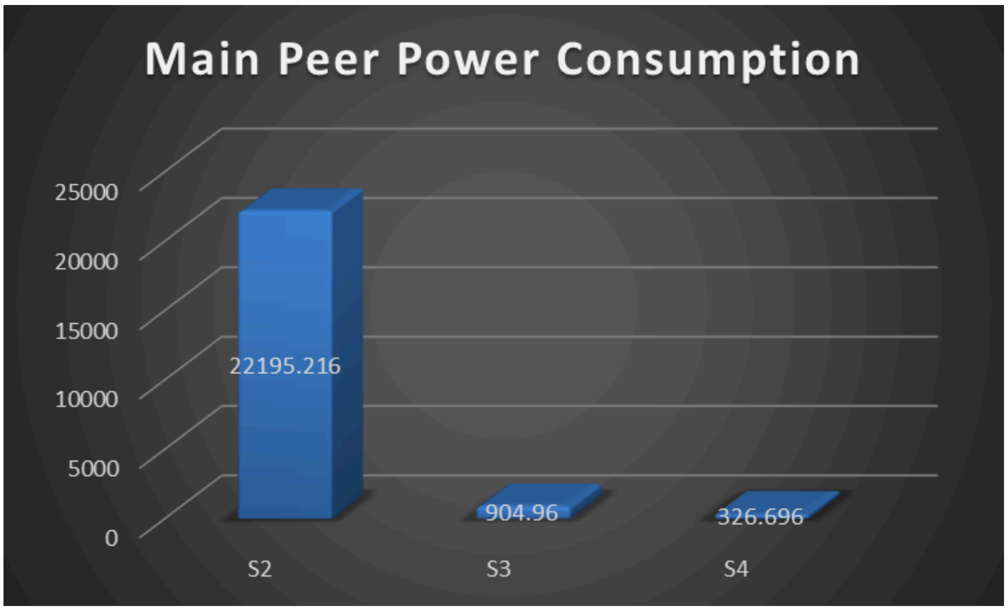


Figure 22: Three Scenarios Main Peer Power Comparison (mJ)

This gradual increase in performance, in turn, is then reflected in the total amount of power consumption within the main peer. By adding and improving these mitigation strategies, the main peer's power consumption is decreased tremendously.

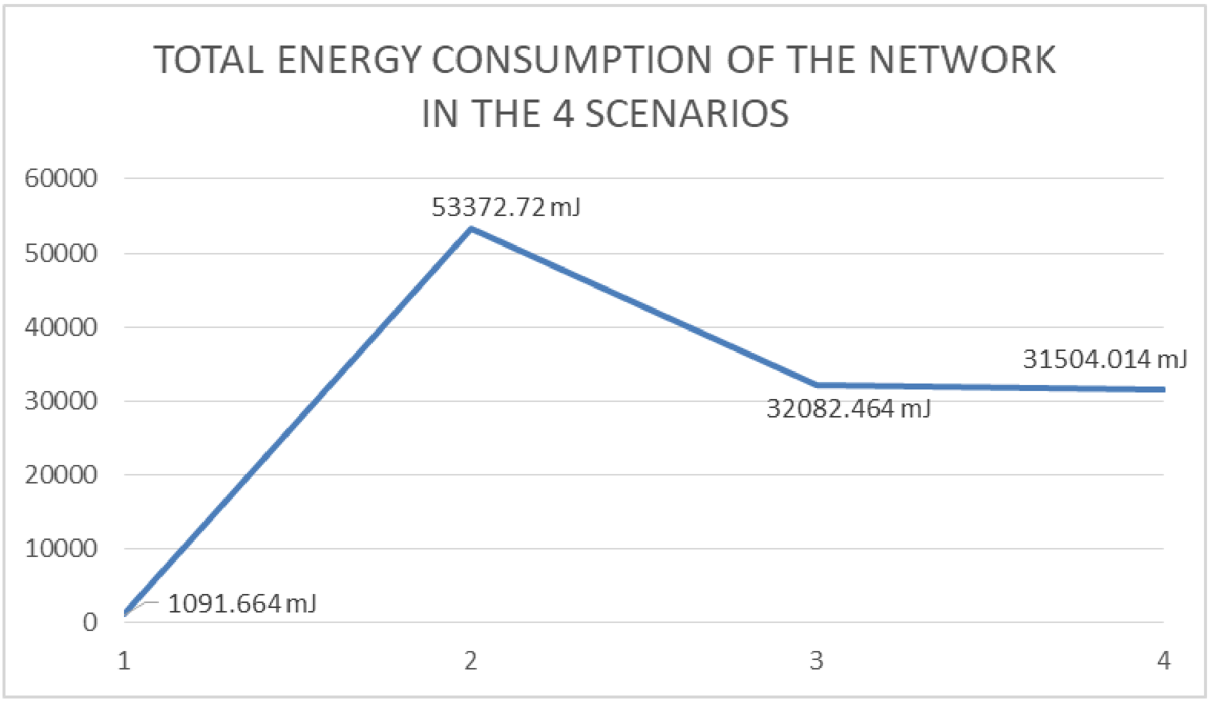


Figure 23: Three Scenarios Main Peer Power Comparison

Overall, by observing the flow of power of the entire network from all scenarios, we again see a gradual decrease in total power consumption. We can see in scenario one that before the DDoS attack, the system's power consumption is minimal, and when the attack is introduced, strain is put on the network. We can then see that when the two mitigation strategies are implemented, the partial amounts of this strain are elevated. This doesn't result in a complete removal of the added power consumption, as the mitigation effects are a reactionary security measure that acts on an existing problem and doesn't prevent the attack before it occurs.

Average attack packet rate during scenario 2	**Average attack packet rate during scenario 3**	**Average attack packet rate during scenario 4**
0.81287982mbps	0.0263957mbps	0.0052896mbps

Table 6: Power analysis

In regards to performance, across the time of attack, we can see the rate at which the main peer receives malicious nodes decreases as we improve the security mitigation metric. This shows that mitigation can slow down the effect of the attack on the main peer, freeing up its processing capabilities.

4. DISCUSSION AND CONCLUSIONS

4.1 Conclusion

Due to the increasing popularity of peer-to-peer networks, it is essential to anticipate and prepare for potential attacks targeting these systems. The study presented in this paper evaluated the energy consumption and performance impact of two mitigation techniques against a DNS Amplification DDoS attack, specifically targeting the peer responsible for the highest volume of traffic. The mitigation methods tested included a timed-rate limiter—applied at the victim to drop packets after a threshold rate from a single source—and a blacklist technique, which permanently blocks packets from senders placed on a peer's internal blacklist.

After conducting multiple simulations and performance comparisons, results indicate that the blacklist mitigation technique was the most effective in this scenario. As shown in Figure 23, Scenario 4 (the blacklist scenario) led to the lowest number of bytes received by the main peer, as well as the lowest overall energy consumption, in comparison to Scenarios 2 and 3.

Although Scenario 3 did not achieve the best overall performance, the timed-rate limiter demonstrated significant mitigation potential. The number of received packets and the energy consumption of the main peer both decreased substantially compared to the unmitigated case. These findings suggest that the timed-rate limiter remains a viable and effective approach for mitigating DDoS attacks in peer-to-peer networks.

4.2 Difficulties

One of the biggest challenges faced in this study was performing the peer-to-peer network simulation, as this type of simulation is not easily supported by NS3 modules as with other software [13]. Multiple functions had to be created to define the nodes' behavior when transmitting data. Also, several logs were needed to make sure the network was working as intended and behaving like a real P2P network.

A second major challenge was implementing spoofing. At the beginning, the DNS resolver was registering queries as coming from the attacker's IP address, which was incorrect, since in a real-world attack, the traffic should not reflect back to the attacker. To fix this, the SpoofHeader class was created, although it was not easy to implement at first. In order to change the attacker's IP address, this class had to inherit from the Header class and replace the original header with the SpoofHeader.

4.3 Future Work

While the simulations successfully modelled a peer-to-peer network under a DNS Amplification DDoS attack and implemented potential mitigation strategies, further improvements are necessary. Future work should incorporate simultaneous packet transmissions between peers, which is a key characteristic of real-world peer-to-peer communication. Additionally, the simulation could be enhanced by introducing a neighbour list for each peer, as data is often preferentially transmitted to physically closer nodes. Across all scenarios, the number of packets sent by the main peer remained relatively constant. Since a primary goal of DDoS attacks is to prevent a node from managing network traffic, future simulations should address and replicate this behaviour more accurately to better understand its impact.

5. REFERENCES

[1] R. Al-Sayyed and A. Ouda, “A survey of P2P overlays in various networks,” Proc. Int. Conf. Inf. Sci. Appl. (ICISA), 2011, pp. 1–7. doi: 10.1109/ICISA.2011.6024559.

[2] N. Singh and S. De, “Survey on recent DDoS mitigation techniques and comparative analysis,” Proc. Int. Conf. Adv. Comput. Commun. Autom. (ICACCA), 2016, pp. 1–5. doi: 10.1109/ICACCA.2016.7546581.

[3] S. T. Vuong and Y. H. Wang, “Distinguishing the master to defend DDoS attack in peer-to-peer networks,” Proc. IEEE Int. Conf. Comput. Intell. Security (CIS), 2010, pp. 212–217. doi: 10.1109/CIS.2010.5578493.

[4] ‘Peer-To-Peer Networks: Features, Pros, and Cons - Spiceworks’, Spiceworks Inc. Accessed: Apr. 16, 2025. [Online]. Available: https://www.spiceworks.com/tech/networking/articles/what-is-peer-to-peer/

[5] ‘Peer-to-peer’, Wikipedia. Feb. 03, 2025. Accessed: Apr. 18, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Peer-to-peer&oldid=1273753861

[6] A. K. Verma and D. Mukhopadhyay, “DDoS testbed based on peer-to-peer grid,” Proc. Int. Conf. Inf. Process. (ICIP), 2017, pp. 1–6. doi: 10.1109/INFOP.2017.7955627.

[7] ‘Top 20 Most Common Types Of Cyber Attacks’, Fortinet. Accessed: Apr. 17, 2025. [Online]. Available: https://www.fortinet.com/resources/cyberglossary/types-of-cyber-attacks

[8] IASbaba, ‘Cyber Attacks in India’, IASbaba. Accessed: Apr. 18, 2025. [Online]. Available: https://iasbaba.com/2022/12/cyber-attacks-in-india/

[9] ‘DDoS threat report for 2023 Q4’, The Cloudflare Blog. Accessed: Apr. 17, 2025. [Online]. Available: https://blog.cloudflare.com/ddos-threat-report-2023-q4/

[10] ‘What is a DDoS Attack: Types, Prevention & Remediation | OneLogin’. Accessed: Apr. 18, 2025. [Online]. Available: https://www.onelogin.com/learn/ddos-attack

[11] V. P. Kumar, A. Sundaram.P, M. B. Kumar, and N.Ch.S.N.Iyengar, ‘ANALYSIS OF DDoS ATTACKS IN DISTRIBUTED PEER TO PEER NETWORKS’, Journal of Global Research in Computer Sciences, vol. 2, no. 6, pp. 10–16, Jan. 1970, Accessed: Apr. 18, 2025. [Online]. Available: https://www.rroij.com/peer-reviewed/analysis-of-ddos-attacks-in-distributed-peer-to-peer-networks-37554.html

[12] ‘Secure Access Service Edge (SASE)’, Workteam. Accessed: Apr. 18, 2025. [Online]. Available: https://workteam.it/en/prodotti/f5-distributed-cloud/

[13] N. Fan, D. L. Goeckel, D. Towsley, and P. Basu, “Simulation of DDoS attacks on P2P networks,” Proc. IEEE Mil. Commun. Conf. (MILCOM), 2011, pp. 1883–1888. doi: 10.1109/MILCOM.2011.6063048.

6. CONTRIBUTIONS

Below outlines the contribution of the team members in the sections outlined in the table:

	Caleb Kuitenbrouwer	Fernando Arias	Clarence Wasilwa
References and literature review	33%	33%	33%
Project website	33%	33%	33%
Simulation scenarios, implementation, analysis, and discussion of simulation results	33%	33%	33%
Project presentation	33%	33%	33%
Written final report	33%	33%	33%

Table 7: Contributions

7. APPENDIX

Peer-to-Peer Simulation Code Explanation:

1. Network and Node Initialization

- Node creation
    - Peers: 6 nodes (0–5) act as P2P clients, 5 normal peers, and 1 main peer.
    - Attacker: node 6 launches the spoofing attack.
    - DNS Resolver: node 7 simulate DNS resolver.
- CSMA channel
    - All eight nodes share a 100 Mbps, 1 ms CSMA link.
- IP addressing & mobility
    - Nodes get IPs in 10.1.1.0/24 and are placed in a 3×3 grid with fixed positions for netAnim animation.

2. Queue Configuration

- Peers' queues are limited to 10 packets to simulate constrained resources.
- The attacker and DNS Resolver each receive larger queues (1,000 packets) to enable them to flood and amplify traffic without local drops.

3. Socket Setup & Role Assignment

- Each peer creates a UDP socket bound to port 8080 and registers the PacketReceived callback.
- clientNames[socket] labels each socket as "Main Peer" (node 0) or "Client i" (nodes 1–5).
- Separate sockets are created for the Attacker and DNS Resolver, also bound to port 8080 and named accordingly.

4. Normal P2P Traffic Generation

- StartSending(socket, destList, port)

    - Draws a random delay: peers use 0–0.05 s, but the "Main Peer" uses a tighter 0–0.01 s window.
    - Schedules one SendPacket to each of its neighbors after that delay.
    - Re‑schedules itself every 5 s, creating continual churn.
- SendPacket(socket, dest, port)
    - Crafts a 1 KiB data packet, sends it via UDP, and increments packetsSentPerNode[node].
    - Logs the send event with timestamp, sender name, and destination.

5. Monitoring, Visualization, and Teardown

- PrintPacketStatistics(...) prints per‑node sent/received/dropped counts at the end.

- NetAnim writes p2p_netanim.xml, labeling each node (Main Peer, Clients, Attacker, DNS Resolver) for later animation.
- The simulation runs for 10 s (Simulator::Run()), then cleans up.

DDoS Simulation Code Explanation:

1. Attack Topology

- Attacker (node 6): issues spoofed DNS queries.
- DNS Resolver (node 7): responds with "amplified" packets.
- Main Peer (node 0): defined as the victim, handling the most "normal" P2P traffic.

All three share the same CSMA channel and UDP port 8080 as the rest of the overlay.

2. Crafting the Spoofed Packet

- SpoofHeader
  - A custom 4‑byte header carrying a fake source IP.
  - Serializes/deserializes that 32‑bit address into each query packet.
- Purpose: lets the attacker make the DNS Resolver believe the query came from the Main Peer.

3. Sending a Spoofed DNS Query

1. Build a 60 Bytes query, stamp on SpoofHeader(mainPeerIp).
2. SendTo DNS Resolver's IP on port 8080.
3. Log the send event for traceability.
4. Simulate resolver behavior:
   - Remove the header, extract spoofedAddress.
   - Create an 8 KiB amplified response and send it back to that spoofed IP.
   - Log the amplified‑response event.

4. Sustaining the Flood

- Immediately calls SendSpoofedDNSQuery.
- Reschedules itself every 0.0065 s (~153 queries per second), creating a continuous flood of amplified packets toward the Main Peer.

5. Victim‑Side Mitigation

Inside PacketReceived(...), the Main Peer runs a simple rate‑limit:

1. Per‑second counter reset
   - When more than one second elapses, packetCountMap is cleared.
2. Counting incoming packets per source IP
   - If any sender exceeds 3 packets/s, its IP is added to blacklist.
3. Blacklist behavior
   - Further packets from that IP are dropped immediately and logged as dropped.

This mimics an elementary threshold‑based DDoS defense.

6. Logging and Metrics

- NS_LOG_WARN for blacklisting events and amplified‑response sends.
- packetsDroppedPerNode increments whenever a packet is dropped due to blacklisting.
- At simulation end, PrintPacketStatistics(...) reports total sent, received, and dropped per node, letting you quantify the attack's impact and the defense's effectiveness.

7. Integration with P2P Overlay

- The attack begins at 10 s into the 10 s run (so it effectively fires just before shutdown), illustrating the system's response under flood conditions.
- Because the P2P peers, attacker, and DNS resolver share the same channel, you can observe how normal overlay traffic degrades, how quickly the Main Peer blacklists sources, and where packets are lost in the network.

8. CODE LISTING

Scenario 2

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/csma-module.h"
#include "ns3/applications-module.h"
#include "ns3/mobility-module.h"
#include "ns3/netanim-module.h"
#include "ns3/random-variable-stream.h"
#include "ns3/flow-monitor-module.h"

#include <map>
#include <set>
#include <vector>
#include <sstream>
#include <algorithm>

using namespace ns3;
std::map<Ptr<Node>, uint32_t> packetsSentPerNode;
std::map<Ptr<Node>, uint32_t> packetsReceivedPerNode;
std::map<Ptr<Node>, uint32_t> packetsDroppedPerNode;

// Define the logging component.
NS_LOG_COMPONENT_DEFINE("P2PNetwork");

/**
 * Custom header to simulate IP spoofing.
 */
class SpoofHeader : public Header {
public:
  SpoofHeader() {}
  void SetSpoofedAddress(Ipv4Address address) { m_spoofedAddress = address; }
  Ipv4Address GetSpoofedAddress() const { return m_spoofedAddress; }

  static TypeId GetTypeId(void) {
    static TypeId tid = TypeId("SpoofHeader")
      .SetParent<Header>()
      .AddConstructor<SpoofHeader>();
    return tid;
  }
  virtual TypeId GetInstanceTypeId(void) const { return GetTypeId(); }
  virtual void Serialize(Buffer::Iterator start) const {
    start.WriteU32(m_spoofedAddress.Get());
  }
  virtual uint32_t Deserialize(Buffer::Iterator start) {
    m_spoofedAddress.Set(start.ReadU32());
    return GetSerializedSize();
  }
  virtual uint32_t GetSerializedSize(void) const { return 4; }
  virtual void Print(std::ostream &os) const { os << "SpoofedAddress=" << m_spoofedAddress; }
private:
  Ipv4Address m_spoofedAddress;
};

// Global maps for naming sockets and tracking ACK suppression.
std::map<Ptr<Socket>, std::string> clientNames;
std::set<std::pair<Ipv4Address, Ipv4Address>> respondedPairs;
const uint32_t blacklistThreshold = 3;
std::map<Ipv4Address, uint32_t> packetCountMap;
std::set<Ipv4Address> blacklist;
Time lastResetTime = Seconds(0);

/**
 * Clears the ACK-suppression state periodically.
 */
void ClearRespondedPairs()
{
  respondedPairs.clear();
  NS_LOG_INFO("[Time: " << Simulator::Now().GetSeconds()
              << "s] Clearing ACK suppression state for next round");
  Simulator::Schedule(Seconds(5.0), &ClearRespondedPairs);
}

/**
 * Callback when a packet is received on any socket.
 */
void PacketReceived(Ptr<Socket> socket)
{
  Address from;
  Ptr<Packet> packet = socket->RecvFrom(from);
  Ipv4Address senderAddress = InetSocketAddress::ConvertFrom(from).GetIpv4();
  std::string name = clientNames[socket];
  Time timestamp = Simulator::Now();


  // Handle ACKs: if packet is 3 bytes, treat as ACK.
  if (packet->GetSize() == 3) {
    NS_LOG_INFO("[Time: " << timestamp.GetSeconds() << "s] "
                << name << " received ACK from " << senderAddress);
    return;
  }
  packetsReceivedPerNode[socket->GetNode()]++;

  NS_LOG_INFO("[Time: " << timestamp.GetSeconds() << "s] "
              << name << " received data packet from " << senderAddress);

  Ptr<Ipv4> ipv4 = socket->GetNode()->GetObject<Ipv4>();
  uint32_t iface = (ipv4->GetNInterfaces() > 1 ? 1 : 0);
  Ipv4Address localAddress = ipv4->GetAddress(iface, 0).GetLocal();

  std::pair<Ipv4Address, Ipv4Address> addrPair(senderAddress, localAddress);
  if (respondedPairs.find(addrPair) == respondedPairs.end()) {
    Ptr<Packet> ackPacket = Create<Packet>((uint8_t*)"ACK", 3);
    socket->SendTo(ackPacket, 0, InetSocketAddress(senderAddress, 8080));
    respondedPairs.insert(addrPair);
    NS_LOG_INFO("[Time: " << timestamp.GetSeconds() << "s] "
                << name << " sent ACK to " << senderAddress);
  }
}

/**
 * Sends a 1024-byte data packet to a destination.
 */
void SendPacket(Ptr<Socket> socket, Ipv4Address destAddress, uint16_t port)
{
  Ptr<Packet> packet = Create<Packet>(1024);
  socket->SendTo(packet, 0, InetSocketAddress(destAddress, port));
  packetsSentPerNode[socket->GetNode()]++;

  NS_LOG_INFO("[Time: " << Simulator::Now().GetSeconds() << "s] "
              << clientNames[socket] << " sent data packet to " << destAddress);
}
```

```
/**
 * Periodically sends packets from a peer.
 * The Main Peer sends more frequently.
 */
void StartSending(Ptr<Socket> socket, std::vector<Ipv4Address> destAddresses, uint16_t port)
{
  Ptr<UniformRandomVariable> uv = CreateObject<UniformRandomVariable>();
  double delay = (clientNames[socket] == "Main Peer") ? uv->GetValue(0.0, 0.01) : uv->GetValue(0.0,
0.05);
  for (auto dest : destAddresses)
  {
    Simulator::Schedule(Seconds(delay), &SendPacket, socket, dest, port);
  }
  Simulator::Schedule(Seconds(5.0), &StartSending, socket, destAddresses, port);
}

/**
 * Sends a spoofed DNS query from the attacker.
 * Attaches a SpoofHeader carrying the Main Peer's IP.
 * The DNS Resolver extracts the spoofed address and sends an amplified response.
 */
void SendSpoofedDNSQuery(Ptr<Socket> attackerSocket, Ptr<Socket> dnsSocket, Ipv4Address
mainPeerAddress, uint16_t port)
{
  Ptr<Packet> query = Create<Packet>(60);
  // Attach the spoof header.
  SpoofHeader spoofHeader;
  spoofHeader.SetSpoofedAddress(mainPeerAddress);
  query->AddHeader(spoofHeader);

  InetSocketAddress dnsResolverAddr =
InetSocketAddress(dnsSocket->GetNode()->GetObject<Ipv4>()->GetAddress(1,0).GetLocal(), port);

  Ptr<Node> attackerNode = attackerSocket->GetNode();
  packetsSentPerNode[attackerNode]++;

  attackerSocket->SendTo(query, 0, dnsResolverAddr);
  NS_LOG_INFO("[Time: " << Simulator::Now().GetSeconds() << "s] Attacker sent spoofed DNS query
(60 bytes + header) to DNS Resolver, spoofing Main Peer (" << mainPeerAddress << ")");

  // Simulate DNS Resolver processing:
  SpoofHeader receivedSpoof;
 query->RemoveHeader(receivedSpoof);
 Ipv4Address spoofedAddress = receivedSpoof.GetSpoofedAddress();

 Ptr<Packet> amplifiedResponse = Create<Packet>(8000);
 Ptr<Node> dnsNode = dnsSocket->GetNode();
 packetsSentPerNode[dnsNode]++;
 dnsSocket->SendTo(amplifiedResponse, 0, InetSocketAddress(spoofedAddress, port));
 NS_LOG_WARN("[Time: " << Simulator::Now().GetSeconds() << "s] DNS Resolver sent amplified
response (6000 bytes) to Main Peer (" << spoofedAddress << ")");
}

/**
 * Repeatedly sends spoofed DNS queries to saturate the Main Peer.
 */
void StartMultipleAttacks(Ptr<Socket> attackerSocket, Ptr<Socket> dnsSocket, Ipv4Address
mainPeerAddress, uint16_t port)
{
  if (Simulator::Now().GetSeconds() > 13.0) return;
 SendSpoofedDNSQuery(attackerSocket, dnsSocket, mainPeerAddress, port);
 Simulator::Schedule(Seconds(0.0065), &StartMultipleAttacks, attackerSocket, dnsSocket,
mainPeerAddress, port);
}
void PrintPacketStatistics(NodeContainer& nodes, uint32_t numPeers, uint32_t totalNodes) {
   std::cout << "\n==== Packet Statistics ====\n";
   for (uint32_t i = 0; i < nodes.GetN(); ++i) {
     Ptr<Node> node = nodes.Get(i);

     std::string name;
     if (i == numPeers)
       name = "Attacker";
     else if (i == totalNodes - 1)
       name = "DNS Resolver";
     else if (i == 0)
       name = "Main Peer";
     else
       name = "Client " + std::to_string(i);

     std::cout << name << ":\n";
     std::cout << "  Packets Sent:     " << packetsSentPerNode[node] << "\n";
     std::cout << "  Packets Received: " << packetsReceivedPerNode[node] << "\n";
     std::cout << "  Packets Dropped:  " << packetsDroppedPerNode[node] << "\n";
   }
 }
int main(int argc, char *argv[])
{
 LogComponentEnable("P2PNetwork", LOG_LEVEL_INFO);

 // Define the network:
 // 6 peers (nodes 0 to 5), 1 Attacker (node 6), 1 DNS Resolver (node 7) = 8 nodes total.
 uint32_t numPeers = 6;
 uint32_t totalNodes = numPeers + 2;
 uint16_t port = 8080;
 double simulationTime = 20.0;
 uint32_t neighborsPerNode = std::min<uint32_t>(3, numPeers - 1);

 // Create nodes.
 NodeContainer nodes;
 nodes.Create(totalNodes);

 CsmaHelper csma;
 csma.SetChannelAttribute("DataRate", StringValue("100Mbps"));
 csma.SetChannelAttribute("Delay", TimeValue(MilliSeconds(1)));
 NetDeviceContainer devices = csma.Install(nodes);
 InternetStackHelper().Install(nodes);

 Ipv4InterfaceContainer interfaces;
 Ipv4AddressHelper ipv4;
 ipv4.SetBase("10.1.1.0", "255.255.255.0");
 interfaces = ipv4.Assign(devices);

 MobilityHelper mobility;
 mobility.SetPositionAllocator("ns3::GridPositionAllocator",
                               "MinX", DoubleValue(0.0),
                               "MinY", DoubleValue(0.0),
                               "DeltaX", DoubleValue(100.0),
                               "DeltaY", DoubleValue(100.0),
                               "GridWidth", UintegerValue(3),
                               "LayoutType", StringValue("RowFirst"));
 mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
 mobility.Install(nodes);

 // Configure the buffer size for all devices.
 // Limit peers to 10 packets; attacker and DNS Resolver get a large queue.
 for (uint32_t i = 0; i < devices.GetN(); i++) {
  Ptr<CsmaNetDevice> device = DynamicCast<CsmaNetDevice>(devices.Get(i));
  if (device) {
    Ptr<Node> node = device->GetNode();
    uint32_t nodeId = node->GetId();
    Ptr<Queue<Packet>> queue = device->GetQueue();
    if (nodeId < numPeers) {
      queue->SetMaxSize(QueueSize("10p"));
    } else {
      queue->SetMaxSize(QueueSize("1000p"));
    }
  }
}

// Create sockets for peers (nodes 0 to 5) and assign roles.
std::vector<Ptr<Socket>> peerSockets(numPeers);
std::vector<Ipv4Address> peerAddresses(numPeers);
for (uint32_t i = 0; i < numPeers; ++i) {
  Ptr<Socket> s = Socket::CreateSocket(nodes.Get(i), UdpSocketFactory::GetTypeId());
  s->Bind(InetSocketAddress(Ipv4Address::GetAny(), port));
  s->SetRecvCallback(MakeCallback(&PacketReceived));
  peerSockets[i] = s;
  if (i == 0)
    clientNames[s] = "Main Peer";
  else
    clientNames[s] = "Client " + std::to_string(i + 1);
}

// Save peer IP addresses.
for (uint32_t i = 0; i < numPeers; ++i) {
  Ptr<Ipv4> ip = nodes.Get(i)->GetObject<Ipv4>();
  peerAddresses[i] = ip->GetAddress((ip->GetNInterfaces() > 1 ? 1 : 0), 0).GetLocal();
}

// Build destination lists for peers.
std::vector<std::vector<Ipv4Address>> destAddressesForPeer(numPeers);
for (uint32_t i = 0; i < numPeers; ++i) {
  for (uint32_t j = 1; j <= neighborsPerNode; ++j)
    destAddressesForPeer[i].push_back(peerAddresses[(i + j) % numPeers]);
}

// Schedule normal traffic for each peer.
for (uint32_t i = 0; i < numPeers; ++i)
```

```
    Simulator::Schedule(Seconds(1.0), &StartSending, peerSockets[i], destAddressesForPeer[i], port);

  // Create the Attacker socket (node 6).
  Ptr<Socket> attackerSocket = Socket::CreateSocket(nodes.Get(numPeers),
UdpSocketFactory::GetTypeId());
  attackerSocket->Bind(InetSocketAddress(Ipv4Address::GetAny(), port));
  attackerSocket->SetRecvCallback(MakeCallback(&PacketReceived));
  clientNames[attackerSocket] = "Attacker";

  // Create the DNS Resolver socket (node 7).
  Ptr<Socket> dnsSocket = Socket::CreateSocket(nodes.Get(totalNodes - 1),
UdpSocketFactory::GetTypeId());
  dnsSocket->Bind(InetSocketAddress(Ipv4Address::GetAny(), port));
  dnsSocket->SetRecvCallback(MakeCallback(&PacketReceived));
  clientNames[dnsSocket] = "DNS Resolver";

  // Schedule the periodic clearing of ACK state.
  Simulator::Schedule(Seconds(6.0), &ClearRespondedPairs);

  // Schedule repeated spoofed DNS queries (the attack).
  // The attacker spoofs the Main Peer’s address so that the amplified responses flood it.
  Simulator::Schedule(Seconds(10.0), &StartMultipleAttacks, attackerSocket, dnsSocket,
peerAddresses[0], port);

  // Install FlowMonitor on all nodes.
  FlowMonitorHelper flowmon;
  Ptr<FlowMonitor> monitor = flowmon.InstallAll();

  // Create an XML animation file for visualization.
  AnimationInterface anim("p2p_netanim.xml");
  for (uint32_t i = 0; i < numPeers; ++i)
    anim.UpdateNodeDescription(nodes.Get(i), clientNames[peerSockets[i]]);
  anim.UpdateNodeDescription(nodes.Get(numPeers), "Attacker");
  anim.UpdateNodeDescription(nodes.Get(totalNodes - 1), "DNS Resolver");

  Simulator::Stop(Seconds(simulationTime));
  Simulator::Run();

  // --------------
  // FLOWMON STATS
  // --------------
  monitor->CheckForLostPackets();
```

```
std::map<FlowId, FlowMonitor::FlowStats> stats = monitor->GetFlowStats();
NS_LOG_INFO("Number of flows: " << stats.size());
for (auto iter = stats.begin(); iter != stats.end(); ++iter) {
    NS_LOG_INFO("Flow " << iter->first << ": TxBytes = " << iter->second.txBytes
                << ", RxBytes = " << iter->second.rxBytes);
}

// ------------------------------------------------
// Aggregate per-node packet & byte statistics
// ------------------------------------------------
std::map<Ipv4Address, uint32_t> nodeTxPackets;
std::map<Ipv4Address, uint32_t> nodeRxPackets;
std::map<Ipv4Address, uint32_t> nodeTxBytes;
std::map<Ipv4Address, uint32_t> nodeRxBytes;

Ptr<Ipv4FlowClassifier> classifier = DynamicCast<Ipv4FlowClassifier>(flowmon.GetClassifier());
for (auto iter = stats.begin(); iter != stats.end(); ++iter) {
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(iter->first);
    const FlowMonitor::FlowStats &fs = iter->second;
    nodeTxPackets[t.sourceAddress] += fs.txPackets;
    nodeRxPackets[t.destinationAddress] += fs.rxPackets;
    nodeTxBytes[t.sourceAddress] += fs.txBytes;
    nodeRxBytes[t.destinationAddress] += fs.rxBytes;
}

 NS_LOG_INFO("=== Aggregated Byte Statistics per Node ===");
for (auto const &entry : nodeTxBytes) {
    NS_LOG_INFO("Node with IP " << entry.first << " sent " << entry.second << " bytes.");
}
for (auto const &entry : nodeRxBytes) {
    NS_LOG_INFO("Node with IP " << entry.first << " received " << entry.second << " bytes.");
}
NS_LOG_INFO("=== Flow Statistics (only flows with lost packets) ===");
for (auto iter = stats.begin(); iter != stats.end(); ++iter) {
    FlowId flowId = iter->first;
    FlowMonitor::FlowStats flowStats = iter->second;
    if (flowStats.lostPackets > 0) {
        Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(flowId);
        uint64_t lostBytesApprox = (flowStats.txBytes > flowStats.rxBytes)
                                  ? (flowStats.txBytes - flowStats.rxBytes)
                                  : 0;
        double meanDelay = 0.0;
        if (flowStats.rxPackets > 0) {
            meanDelay = flowStats.delaySum.GetSeconds() / flowStats.rxPackets;
        }
        NS_LOG_INFO("Flow " << flowId << " ("
                    << t.sourceAddress << " -> " << t.destinationAddress
                    << ", " << t.sourcePort << " -> " << t.destinationPort
                    << ", protocol=" << (uint16_t)t.protocol << "):"
                    << "\n  TxBytes:      " << flowStats.txBytes
                    << "\n  RxBytes:      " << flowStats.rxBytes
                    << "\n  LostPackets:  " << flowStats.lostPackets
                    << "\n  LostBytes:    " << lostBytesApprox
                    << "\n  DelaySum:     " << flowStats.delaySum.GetSeconds() << " s"
                    << "\n  MeanDelay:    " << meanDelay << " s"
                    << "\n  JitterSum:    " << flowStats.jitterSum.GetSeconds() << " s\n");
    }
}

// ------------------------------------------------
// Additionally, print detailed statistics for target Flow (Flow 32)
// ------------------------------------------------
FlowId targetFlowId = 32;
if (stats.find(targetFlowId) != stats.end()) {
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(targetFlowId);
    FlowMonitor::FlowStats flowStats = stats[targetFlowId];
    double meanDelay = 0.0;
    if (flowStats.rxPackets > 0) {
        meanDelay = flowStats.delaySum.GetSeconds() / flowStats.rxPackets;
    }
    NS_LOG_INFO("Flow " << targetFlowId << " 5-tuple: "
                << t.sourceAddress << " -> " << t.destinationAddress
                << ", " << t.sourcePort << " -> " << t.destinationPort
                << ", protocol: " << (uint16_t)t.protocol);
    uint64_t lostBytesApprox = (flowStats.txBytes > flowStats.rxBytes)
                               ? (flowStats.txBytes - flowStats.rxBytes)
                               : 0;
    NS_LOG_INFO("Flow " << targetFlowId << " stats:"
                << "\n  TxBytes = " << flowStats.txBytes
                << "\n  RxBytes = " << flowStats.rxBytes
                << "\n  LostPackets = " << flowStats.lostPackets
                << "\n  LostBytes   = " << lostBytesApprox
                << "\n  DelaySum    = " << flowStats.delaySum.GetSeconds() << " s"
                << "\n  MeanDelay   = " << meanDelay << " s"
            << "\n  JitterSum   = " << flowStats.jitterSum.GetSeconds() << " s");
 } else {
    NS_LOG_WARN("Flow " << targetFlowId << " not found in FlowMonitor stats.");
 }

 // Optionally, serialize FlowMonitor statistics to an XML file.
 monitor->SerializeToXmlFile("flowmon-results.xml", true, true);
 PrintPacketStatistics(nodes, numPeers, totalNodes);

 Simulator::Destroy();
 return 0;
}
```

Scenario 3 Modification

```
if (name == "Main Peer")
{
    if ((timestamp - lastResetTime).GetSeconds() >= 1.0)
    {
        packetCountMap.clear();
        lastResetTime = timestamp;
    }
    packetCountMap[senderAddress]++;
    //Over 10 from the same user sub second just never receive
    if (packetCountMap[senderAddress] > packetThreshold)
    {
        NS_LOG_WARN("[Mitigation] Main peer dropped the packet sent from " <<
senderAddress
                    << " at time " << timestamp.GetSeconds() << "s as its
behaviour is suspicious");
                 packetsDroppedPerNode[socket->GetNode()]++;

        return;
    }
}
```

Scenario 4 Modification

```
// DDoS mitigation: if sender is blacklisted, drop packet.
 if (blacklist.find(senderAddress) != blacklist.end())
 {
    NS_LOG_WARN("[Blacklist] " << name << " dropped a packet from "
                              << senderAddress << " at " << timestamp.GetSeconds() << "s");
  packetsDroppedPerNode[socket->GetNode()]++;
    return;
}

// For Main Peer, update packet count and maybe blacklist.
if (name == "Main Peer")
{
    if ((timestamp - lastResetTime).GetSeconds() >= 1.0)
    {
        packetCountMap.clear();
        lastResetTime = timestamp;
    }
    packetCountMap[senderAddress]++;
    if (packetCountMap[senderAddress] > blacklistThreshold)
    {
        blacklist.insert(senderAddress);
        NS_LOG_WARN("[Mitigation] " << senderAddress << " blacklisted at "
                                  << timestamp.GetSeconds() << "s");
       packetsDroppedPerNode[socket->GetNode()]++;

        return;
    }
}
```