ENSC 427: COMMUNICATION NETWORKS

Spring 2025

Analyzing the Security of Email Transmission in NS3 Through DDoS Attacks

URL: https://www.sfu.ca/~jbl16/

Team 2:

Joel Lerner	301429649	jbl16@sfu.ca
Xin Wei	301414467	xwa202@sfu.ca
Devon Sandhu	301348929	dss17@sfu.ca

Abstract

Distributed Denial-of-Service (DDoS) attacks pose a significant threat to reliable communication, potentially disrupting access to critical online services. Email transmission, which relies on TCP encryption protocols for security, can be heavily impacted by such attacks. This project aims to analyze the effects of DDoS attacks on secure email transmission using NS3 simulations. The study will model an email communication system between client and server while simulating different DDoS attack scenarios by varying attack intensity and traffic patterns. Performance metrics such as throughput, latency, and packet loss will be evaluated using NS3's FlowMonitor module. The goal of this research is to assess the resilience of secure email systems under DDoS conditions and identify strategies to optimize network performance and mitigate attack impacts.

Introduction

Email plays a vital role in reliable and safe communications for companies and individuals across the globe. It is important that we understand weaknesses associated with email and the potential attack strategies malicious users could employ against them.

The primary objective of this research is to quantify the effects of DDOS attacks on email communication. The study will discuss the results of a simulated DDOS attack on an email communication system using NS3. We will analyse throughput, latency, and packet loss while varying the intensity of the attack. Additionally we will be analysing potential detection and defence mechanisms against DDoS attacks on mail servers.

Understanding how email systems behave under DDOS attack conditions is crucial for designing more resilient communication infrastructures. The findings of this research can assist network administrators, cybersecurity professionals, and email service providers in implementing better security for email transmission.

Literature Overview

How to implement Simple Mail Transfer Protocol in NS3: The code referenced for SMTP client/server setup was found at NS3Simulation.com [1]. The guide contained a comprehensive overview of the necessary setup, as well as sample code which we used to build our implementation on top of.

DDoS Attacks and Defense Mechanisms: A Classification, provided a comprehensive overview of Distributed Denial of Service (DDoS) attacks and corresponding defence strategies [2]. The paper classifies DDoS attacks as being either bandwidth or resource depletion attacks. The paper makes note of the layered nature of DDoS attacks and identifies that effective defence against the attacks requires detection, response and recovery techniques. A large variety of defence strategies are presented which opens the opportunity for analysis of the effectiveness of each through simulation.

Overview of DDoS Attack Detection in Software-Defined Networks: a comprehensive review of detection techniques in SDN environments [3]. It classifies methods into statistical, machine learning, and deep learning approaches, and highlights the vulnerabilities of centralized controllers. The paper also reviews data preprocessing and dimensionality reduction methods, and discusses challenges for future research.

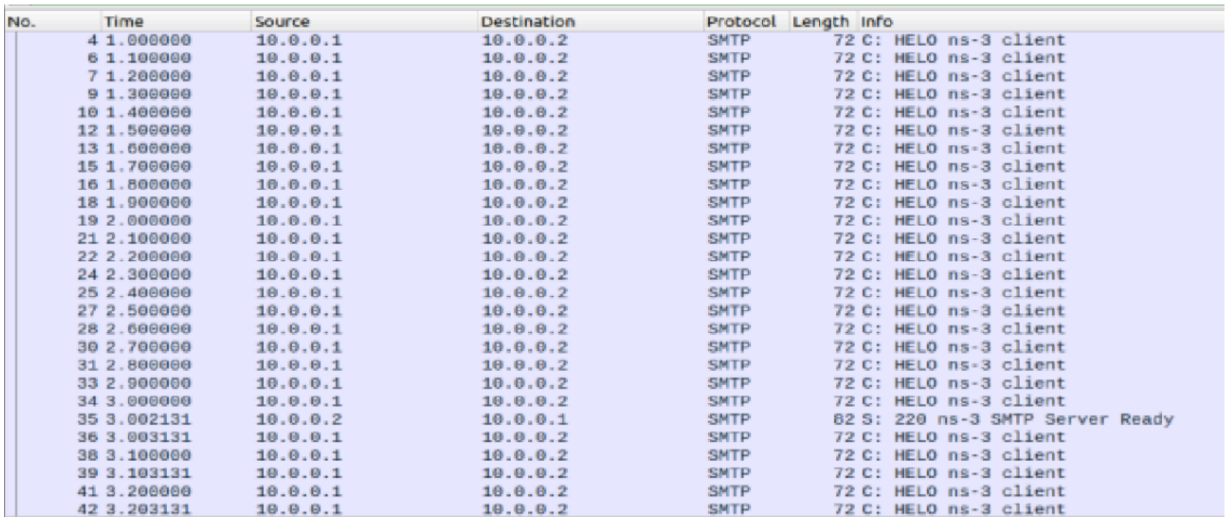
An Effective Mechanism to Mitigate Real-Time DDoS Attack: an optimized SVM-based framework integrated with SNORT IPS for real-time detection and mitigation of DDoS attacks [4]. It demonstrates higher accuracy and faster response compared to traditional methods, underlining the importance of early detection and prompt mitigation.

Distributed Denial of Service (DDoS)

A Distributed Denial of Service (DDoS) attack is a malicious attempt to overwhelm a target service with a flood of traffic rendering the service unavailable to legitimate users [2]. The typical DDoS attack scenario consists of a target, a server for example, and an attacker who controls multiple other client nodes known as “Bot nodes”. The bot nodes are usually machines who’s operators are unaware that they are being used in an attack. Using a large number of bot nodes allows the attacker to send larger volumes of traffic without driving a suspicious amount of traffic from any one single node. Therefore, in general the more nodes an attacker has access to, the more effective the attack will be. A DDoS attack’s effectiveness is typically quantified by the degree to which the service’s bandwidth is occupied and the duration for which the attack is sustained [5].

To protect services from DDoS attacks, a variety of detection and defence mechanisms have been developed. Detection methods include detection by activity and detection by location [2]. In this analysis we will be focusing on activity detection, which involves monitoring traffic on a service and looking for anomalies such as excessive amounts of traffic from a small number of IP addresses indicating that there might be an attack. Once detected, traffic from suspicious nodes can be either refused completely or simply rate limited.

Email servers can be targeted by a DDoS attack via a flood of Simple Mail Transfer Protocol (SMTP) commands sent from bot nodes. SMTP communication between a client and server is initialized using the SMTP ‘HELO’ command from the client, to which the server responds with a’ 220 server ready’ command. A bot node can be used to spam the target server with a high volume of the ‘HELO’ commands forcing the server to try to respond to each command. If the spam volume is large enough, this would put the server into a DDoS state, not having enough bandwidth to respond to any legitimate clients.



No.	Time	Source	Destination	Protocol	Length	Info
4	1.000000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
6	1.100000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
7	1.200000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
9	1.300000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
10	1.400000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
12	1.500000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
13	1.600000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
15	1.700000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
16	1.800000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
18	1.900000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
19	2.000000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
21	2.100000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
22	2.200000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
24	2.300000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
25	2.400000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
27	2.500000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
28	2.600000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
30	2.700000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
31	2.800000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
33	2.900000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
34	3.000000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
35	3.002131	10.0.0.2	10.0.0.1	SMTP	82	S: 220 ns-3 SMTP Server Ready
36	3.003131	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
38	3.100000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
39	3.103131	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
41	3.200000	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client
42	3.203131	10.0.0.1	10.0.0.2	SMTP	72	C: HELO ns-3 client

Figure 1. SMTP packet capture during attack

Implementation

The simulation environment for this project was established using the ns-3 network simulation framework. We developed three distinct ns-3 applications—SMTP Server, SMTP Client, and SMTP Attacker—which were deployed on separate network nodes interconnected by point-to-point links, thus forming a complete experimental topology.

SMTP Server

The SMTP Server application is implemented using a custom ns-3 class named `ns3::SmtpServer`, derived from `ns3::Application`. This application listens for incoming SMTP connections via a TCP socket bound to a specified local address and port (defaulting typically to port 25). Upon initialization in the `StartApplication()` method, the server creates and binds the socket to the configured local address and begins listening for incoming connections. Two essential callbacks are established for handling server-side events: one callback (`HandleAccept()`) manages newly incoming connection requests and maintains active client sockets, while the other callback (`HandleRead()`) processes incoming SMTP requests.

Each received packet triggers the server to respond with a standard SMTP greeting ("220 ns-3 SMTP Server Ready\r\n"), confirming readiness for SMTP communication. Active connections are managed using a vector (`std::vector`) of socket pointers, which stores references to all accepted client connections. When the server application is terminated using the `StopApplication()` method, it ensures all open sockets, including the listening socket and active client connections, are properly closed, and all associated resources are released.

SMTP Client

The SMTP Client application was defined as the class `ns3::SmtpClient`, also subclassing `ns3::Application`. On startup, the client establishes a TCP connection with the SMTP Server's listening address. It initially schedules the first SMTP request message ("HELO ns-3 client\r\n") to be sent after 2.88 milliseconds, ensuring a consistent sending rate of approximately 347 requests per second. Responses from the server are received and logged within the `HandleRead()` callback. If a response is successfully received, the client schedules the next request transmission at the same interval. Upon shutdown, the client closes its connection socket and cancels any pending scheduled transmissions.

SMTP Attacker

For simulating a realistic attack scenario, the SMTP Attacker application was implemented through the `ns3::SmtpAttacker` class, likewise derived from `ns3::Application`. In contrast to the SMTP Client, the attacker uses UDP sockets for high-speed, connectionless packet transmission. The attacker continuously transmits packets containing a fixed payload size of 1400 bytes to the SMTP Server at a high frequency of one packet per millisecond, simulating intense DDoS traffic at approximately 1000 packets per second. Although the attacker's application defines a receive callback for UDP responses, it typically remains inactive as the server does not usually respond to unsolicited UDP traffic. At application termination, the attacker closes its UDP socket and halts all scheduled packet transmissions.

Simulation Topology

The simulation topology itself consists of one client node (green), one server node (blue), one router node (black), and three attacker nodes (red). Each node connects through dedicated point-to-point links characterized by a bandwidth of 5 Mbps and a delay of 2 milliseconds. The network addressing scheme assigns distinct subnets: 10.0.1.0/24 for the client-router link, 10.0.2.0/24 for the server-router link, and 10.0.3.0/24 to 10.0.5.0/24 for the attacker-router connections. ns-3 automatically generates global routing tables to facilitate accurate packet forwarding across the network.

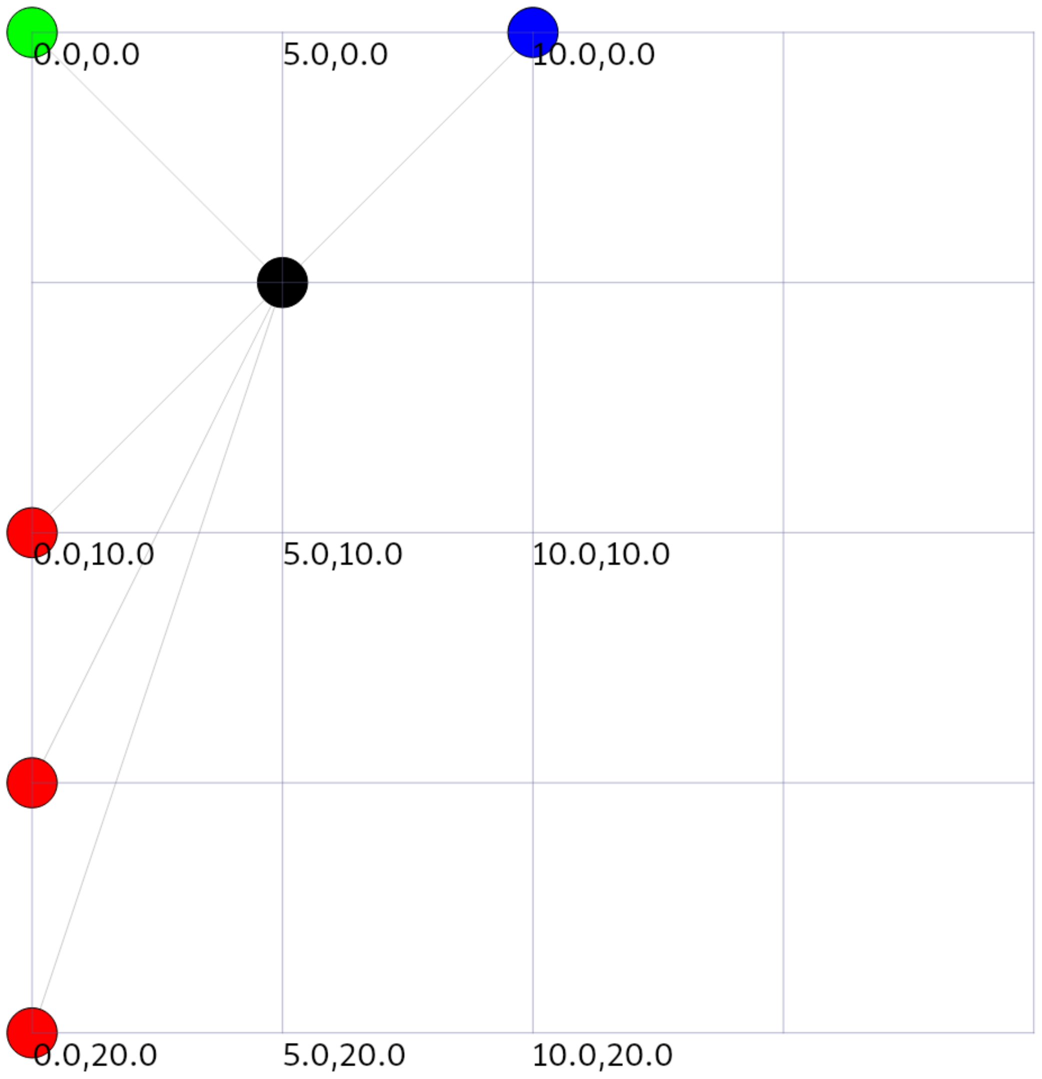
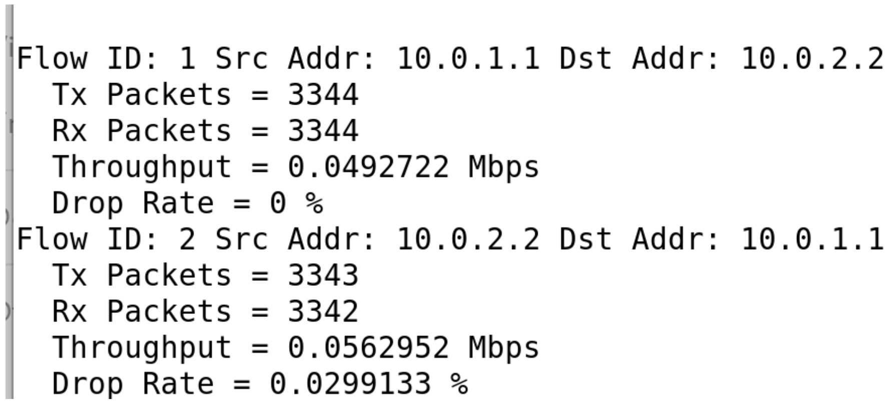


Figure 2: Netanim Visual of Simulation

In terms of application scheduling, the SMTP Server application begins operation at simulation time 1 second and continues until 40 seconds. The SMTP Client starts at 2 seconds and consistently sends SMTP requests until 40 seconds at the specified 2.88 millisecond interval. The SMTP Attacker applications activate at simulation time 4 seconds, generating intensive UDP traffic until the end of the simulation at 40 seconds.

Before introducing malicious behavior, a baseline was established by configuring a legitimate SMTP client to send traffic at a rate of 50 kbps. This was validated using flowmonitor statistics, which can be seen below, giving us a throughput of about 50 kbps from the client (10.0.1.1) to the server (10.0.2.2).

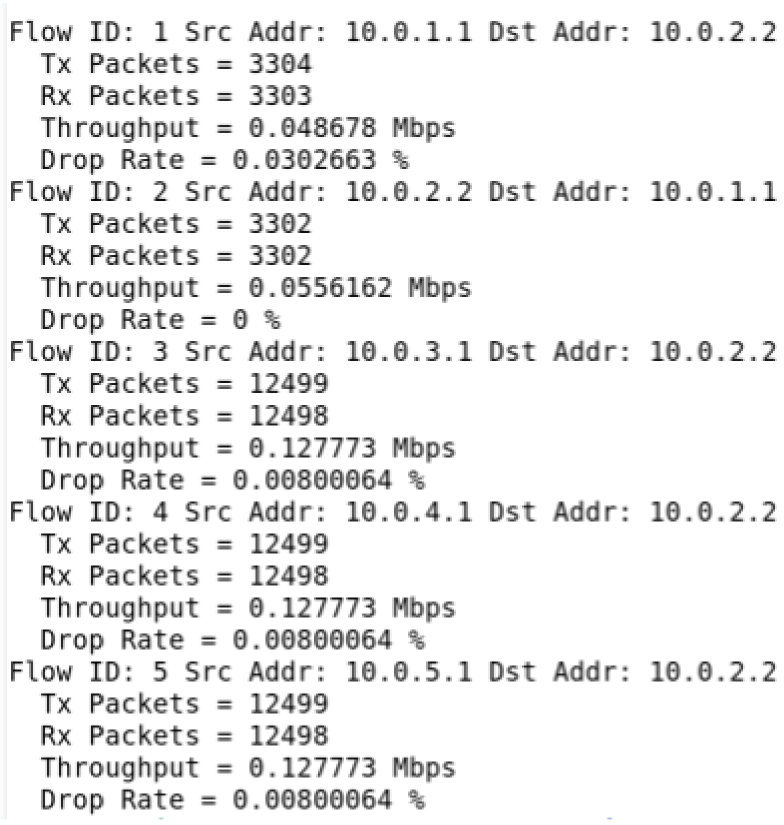


```
Flow ID: 1 Src Addr: 10.0.1.1 Dst Addr: 10.0.2.2
  Tx Packets = 3344
  Rx Packets = 3344
  Throughput = 0.0492722 Mbps
  Drop Rate = 0 %
Flow ID: 2 Src Addr: 10.0.2.2 Dst Addr: 10.0.1.1
  Tx Packets = 3343
  Rx Packets = 3342
  Throughput = 0.0562952 Mbps
  Drop Rate = 0.0299133 %
```

Figure 3: Client Ideal Throughput

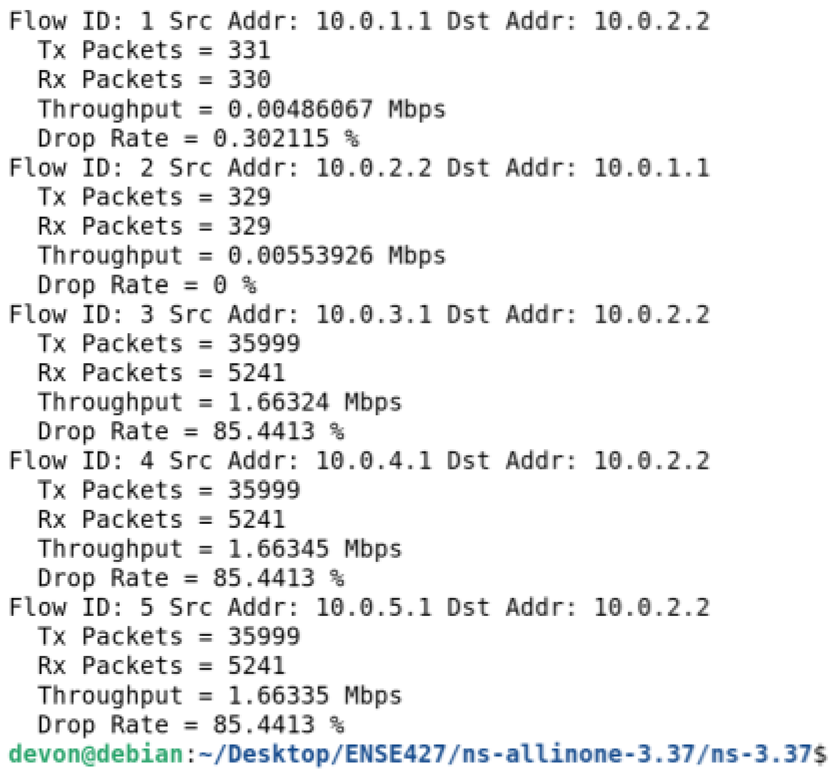
DDoS attacks

For our first attack simulation we set up 3 attack nodes to throttle the link between the router and server with the goal of decreasing client throughput. We varied the rate of the attack from the same rate as the client, 50 kbps, to upwards of 300 Mbps. The figures below show flowmonitor statistics of both a low rate attack, and a high rate attack. With the attackers having addresses of: 10.0.3.1, 10.0.4.1, 10.0.5.1. Later, a different method of attack involving a variable number of attackers was analysed.



```
Flow ID: 1 Src Addr: 10.0.1.1 Dst Addr: 10.0.2.2
  Tx Packets = 3304
  Rx Packets = 3303
  Throughput = 0.048678 Mbps
  Drop Rate = 0.0302663 %
Flow ID: 2 Src Addr: 10.0.2.2 Dst Addr: 10.0.1.1
  Tx Packets = 3302
  Rx Packets = 3302
  Throughput = 0.0556162 Mbps
  Drop Rate = 0 %
Flow ID: 3 Src Addr: 10.0.3.1 Dst Addr: 10.0.2.2
  Tx Packets = 12499
  Rx Packets = 12498
  Throughput = 0.127773 Mbps
  Drop Rate = 0.00800064 %
Flow ID: 4 Src Addr: 10.0.4.1 Dst Addr: 10.0.2.2
  Tx Packets = 12499
  Rx Packets = 12498
  Throughput = 0.127773 Mbps
  Drop Rate = 0.00800064 %
Flow ID: 5 Src Addr: 10.0.5.1 Dst Addr: 10.0.2.2
  Tx Packets = 12499
  Rx Packets = 12498
  Throughput = 0.127773 Mbps
  Drop Rate = 0.00800064 %
```

Figure 4: Low Attack Rate Flowmonitor Statistics



```
Flow ID: 1 Src Addr: 10.0.1.1 Dst Addr: 10.0.2.2
  Tx Packets = 331
  Rx Packets = 330
  Throughput = 0.00486067 Mbps
  Drop Rate = 0.302115 %
Flow ID: 2 Src Addr: 10.0.2.2 Dst Addr: 10.0.1.1
  Tx Packets = 329
  Rx Packets = 329
  Throughput = 0.00553926 Mbps
  Drop Rate = 0 %
Flow ID: 3 Src Addr: 10.0.3.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 5241
  Throughput = 1.66324 Mbps
  Drop Rate = 85.4413 %
Flow ID: 4 Src Addr: 10.0.4.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 5241
  Throughput = 1.66345 Mbps
  Drop Rate = 85.4413 %
Flow ID: 5 Src Addr: 10.0.5.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 5241
  Throughput = 1.66335 Mbps
  Drop Rate = 85.4413 %
devon@debian:~/Desktop/ENSE427/ns-allinone-3.37/ns-3.37$
```

Figure 5: High Attack Rate Flowmonitor Statistics

DDoS detection

To mitigate the impact of a simulated DDoS attack on the SMTP server, a traffic shaping defense was implemented at the router level using the Token Bucket Filter (TBF) queue discipline. The Token Bucket Filter (TBF) operates by controlling packet transmission based on a system of tokens, where tokens represent permission to send a certain number of bytes [6]. Tokens accumulate at a steady rate and are required to send packets, regardless of packet size, to reflect real-world link usage [6]. Initially, the bucket may be full, allowing for a burst of traffic up to a specified limit [6]. If there are insufficient tokens, incoming packets are placed in a queue up to a defined maximum size [6]. Transmission is delayed until enough tokens accumulate to permit sending the first queued packet. To prevent excessive bursts, a peak rate can be enforced.

Ns-3 includes a built-in queue discipline named `ns3::TbfQueueDisc`, in the traffic control module. This regulates packet flow based on a defined transmission rate and buffer capacity. It was applied directly to the router to server link, effectively shaping traffic before it reached the server.

The following parameters were used:

- **Rate**: 5Mbps
  Matches the capacity of the bottleneck link. Prevents the queue from accepting

more packets than the link can handle.

- **Burst**: 1400 (bytes)
  The maximum amount of bytes that “tokens” can be available at one instant

- **MaxSize**: "10000p" (packets)
  Specifies the maximum queue size.

Results

Varying Attack Rate

This section displays results from simulated DDoS attacks with varying attack rates. Three attacker nodes were used to flood the email server with SMTP HELO commands. The total attack rate is swept from 0 to 350Mbps.

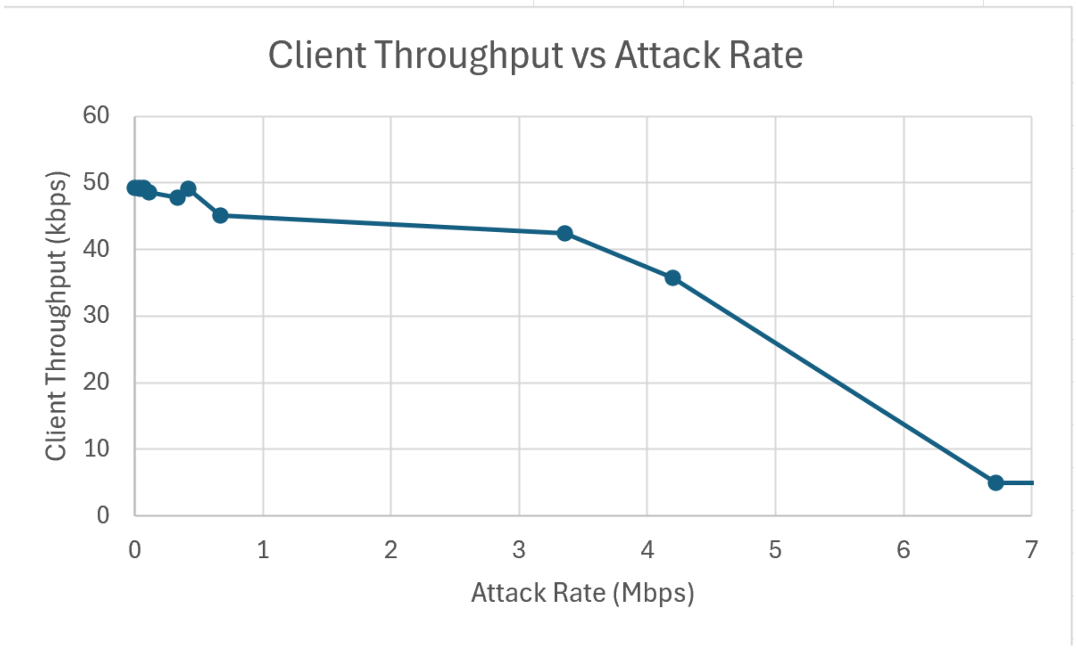


Figure 6: Variable Attack Rate

In figure 6 above we can see that at attack rates below the maximum link rate (5 Mbps), the client is able to maintain a relatively steady throughput of around 50 kbps. However, when the attack rate exceeds the link rate, the client throughput drops rapidly by around 10 times.

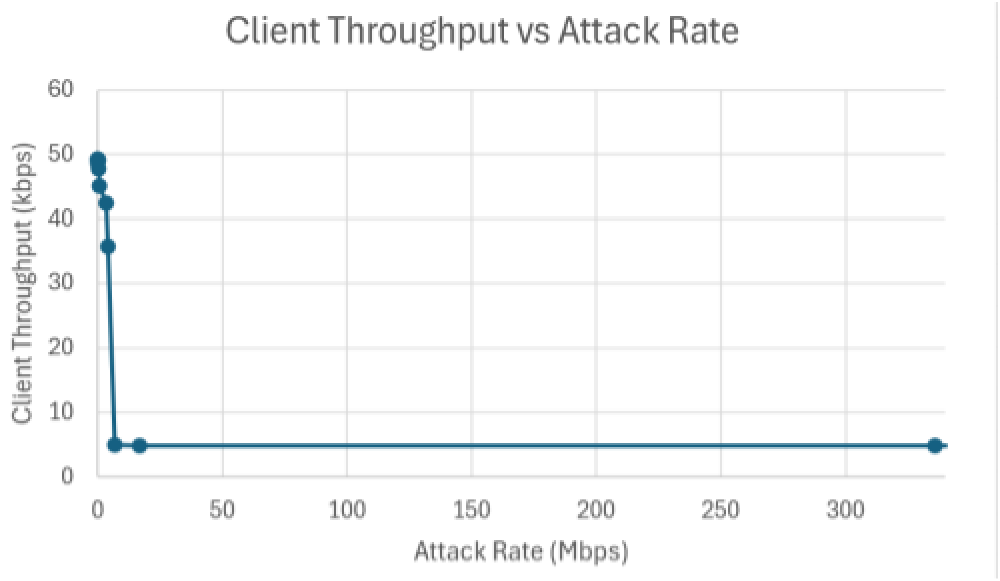


Figure 7: Minimum Client Throughput

Figure 7 shows that after the attack rate reaches the link bandwidth of 5Mbps, increasing the rate of attack does not further decrease the legitimate clients throughput. The client's throughput seems to have hit a minimum of around 4.8 kbps. This is due to the client's successful messages to the server prior to the start of the DDoS attack in each simulation. Therefore, the client is fully denied service after the link bandwidth has been exceeded by the attackers.

Varying Number of Attackers

This section displays results from simulated DDoS attacks with a varying number of attacker nodes. All attackers are sending at the same rate as the legitimate client (50kbps). The number of attackers is swept from 3 to 200.

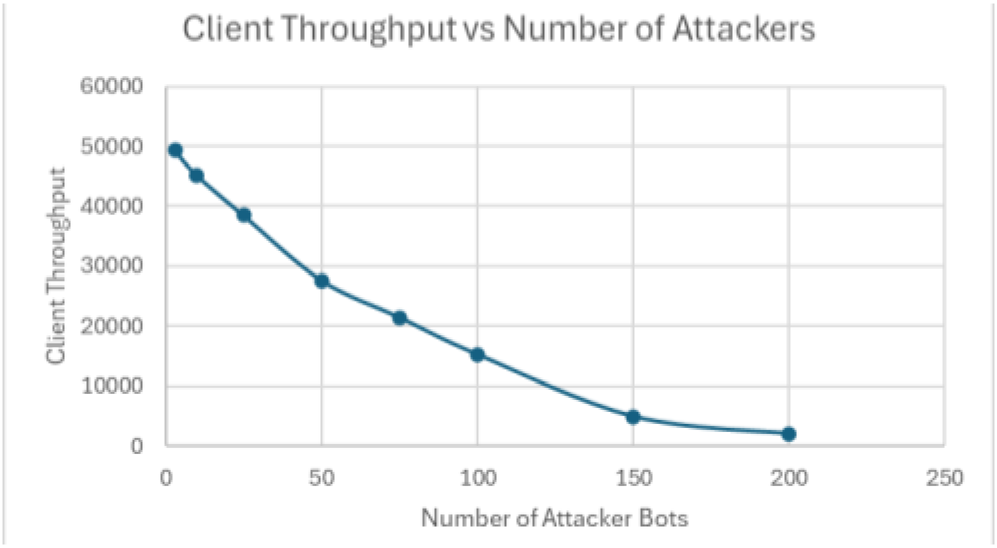
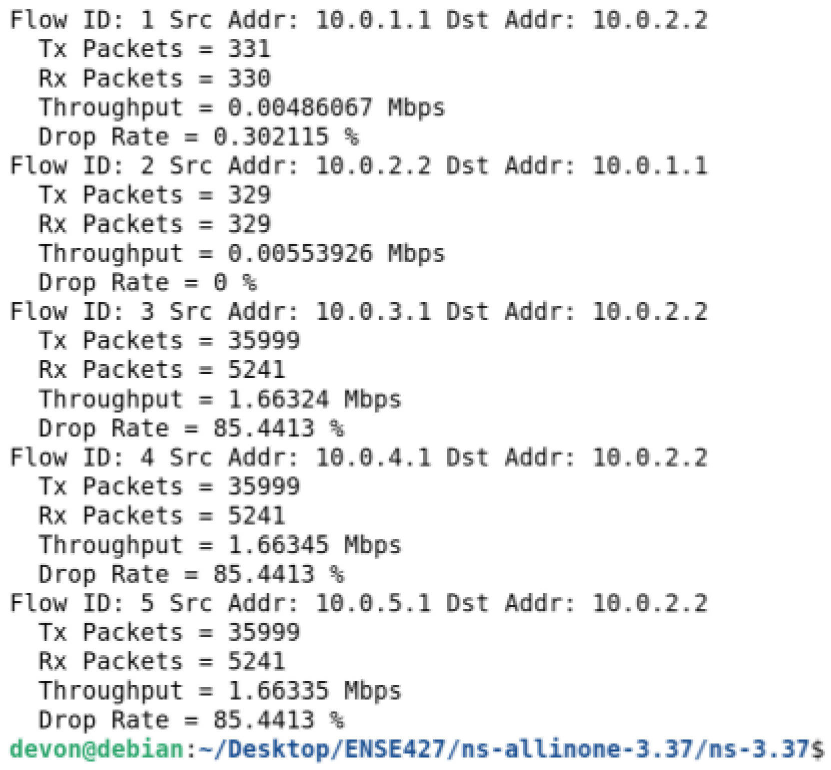


Figure 8: Variable number of Attackers

As seen in figure 8, increasing the number of attackers causes the legitimate client's throughput to decrease as expected. However, the client's throughput begins to drop immediately rather than once the attacker's throughput causes the 5 Mbps bandwidth to be exhausted (this occurs at 100 bots sending at 50kbps). Theoretically, we believe this early decline in throughput to be due to the TCP handshaking that every additional node in the network must complete to establish its connection. When varying the attack rate, we didn't see this phenomenon as the number of attackers was kept constant.

Impact of The Defense

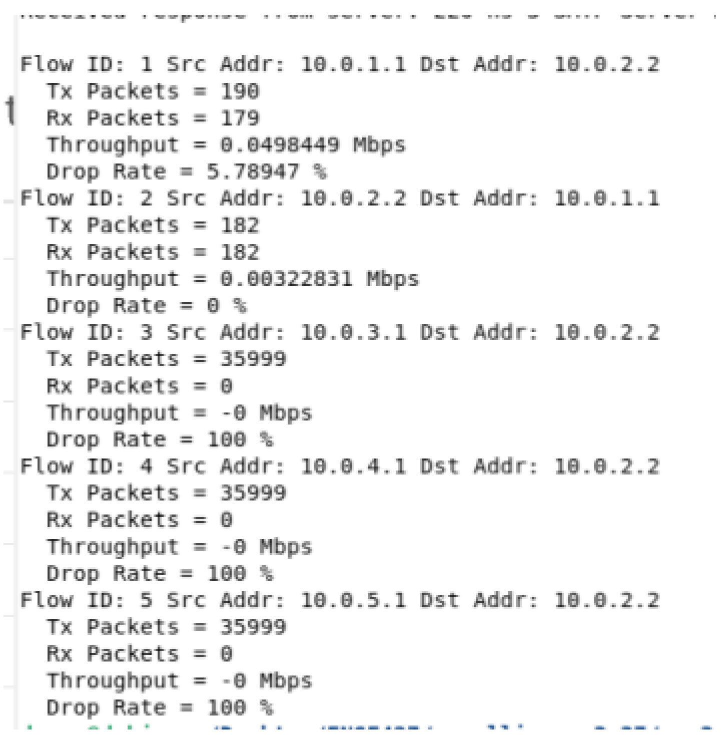
We can see in figure 9 below that with no defense in place the attackers are successful in throttling the link with traffic. This is observed by the throughput of the client being only around 4.8 kbps compared to our desired 50 kbps.



```
Flow ID: 1 Src Addr: 10.0.1.1 Dst Addr: 10.0.2.2
  Tx Packets = 331
  Rx Packets = 330
  Throughput = 0.00486067 Mbps
  Drop Rate = 0.302115 %
Flow ID: 2 Src Addr: 10.0.2.2 Dst Addr: 10.0.1.1
  Tx Packets = 329
  Rx Packets = 329
  Throughput = 0.00553926 Mbps
  Drop Rate = 0 %
Flow ID: 3 Src Addr: 10.0.3.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 5241
  Throughput = 1.66324 Mbps
  Drop Rate = 85.4413 %
Flow ID: 4 Src Addr: 10.0.4.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 5241
  Throughput = 1.66345 Mbps
  Drop Rate = 85.4413 %
Flow ID: 5 Src Addr: 10.0.5.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 5241
  Throughput = 1.66335 Mbps
  Drop Rate = 85.4413 %
devon@debian:~/Desktop/ENSE427/ns-allinone-3.37/ns-3.37$
```

Figure 9: DDOS with No Defense

With TBF in place, the router stops excessive incoming traffic, ensuring that no more than 5 Mbps worth of packets can pass through to the server. Figure 10 shows that while the client did not deliver as many packets, it was able to achieve its set throughput of 50 kbps. Furthermore, we can see that all attacking packets were dropped before reaching the server. The lower number of delivered client packets with TBF in place suggests that legitimate packets were likely being queued or even dropped due to the fixed rate and burst constraints imposed by the TBF. In scenarios of high congestion, client packets may spend excessive time in the queue or exceed the buffer size, preventing timely delivery before the simulation ends.



```
Flow ID: 1 Src Addr: 10.0.1.1 Dst Addr: 10.0.2.2
  Tx Packets = 190
  Rx Packets = 179
  Throughput = 0.0498449 Mbps
  Drop Rate = 5.78947 %
Flow ID: 2 Src Addr: 10.0.2.2 Dst Addr: 10.0.1.1
  Tx Packets = 182
  Rx Packets = 182
  Throughput = 0.00322831 Mbps
  Drop Rate = 0 %
Flow ID: 3 Src Addr: 10.0.3.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 0
  Throughput = -0 Mbps
  Drop Rate = 100 %
Flow ID: 4 Src Addr: 10.0.4.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 0
  Throughput = -0 Mbps
  Drop Rate = 100 %
Flow ID: 5 Src Addr: 10.0.5.1 Dst Addr: 10.0.2.2
  Tx Packets = 35999
  Rx Packets = 0
  Throughput = -0 Mbps
  Drop Rate = 100 %
```

Figure 10: Flowmonitor Statistics after implementing TBF

Future Work

While the current implementation demonstrates a basic traffic shaping method to mitigate DDoS attacks on SMTP servers, detecting these attacks accurately remains a significant challenge. This is primarily due to the increasing sophistication of attackers, who continuously adopt varied and complex methods to evade traditional detection systems. As such, future work should focus on developing and implementing advanced anomaly detection algorithms capable of identifying abnormal traffic patterns with greater accuracy and efficiency. These methods may include statistical analysis, machine learning techniques, or hybrid models designed specifically for real-time detection of various DDoS attack forms.

Moreover, given the diversity of DDoS attack strategies—such as volumetric attacks, protocol exploitation, or low-and-slow attacks—future research should aim to systematically categorize and simulate these distinct attack types within the ns-3 environment. This will enable comprehensive performance evaluation and facilitate the development of tailored defense mechanisms specific to each identified category of attack. Advanced simulation scenarios should include more realistic attacker behaviors and varied traffic patterns, helping researchers and cybersecurity professionals better understand how different forms of DDoS attacks impact SMTP servers and the effectiveness of various mitigation strategies.

Conclusion

This project analyzed the vulnerability of email communication to DDoS attacks using the NS3 framework. Legitimate client throughput was seen to degrade as the volume of spam traffic increased both through increased attack rates and increased number of attackers. Additionally, DDoS detection and filtering was implemented using a TBF queue discipline resulting in the ability to filter out malicious attackers provided the attack rate was large enough.

Our results confirm that SMTP servers are susceptible to a range of DDoS attacks especially attacks with large numbers of attackers. In the case that an attack is not successfully prevented, we see legitimate clients denied service to nearly 0 bps throughput. In the case that the attack is detectable and a filter is applied, we see that the attack can be denied completely resulting in the clients bandwidth remaining un-effected.

These findings highlight the importance of network defenses such as traffic shaping in protecting critical communication infrastructure. As DDoS tactics grow more sophisticated, ongoing work is needed to ensure that email servers remain robust, reliable, and secure in real-world conditions.

References

[1] ns3simulation.com, “How to implement Simple Mail Transfer Protocol in NS3,” *ns3simulation.com*, [Online]. Available: https://ns3simulation.com/how-to-implement-simple-mail-transfer-protocol-in-ns3/. [Accessed: Apr. 17, 2025].

[2] C. Douligeris and A. Mitrokotsa, "DDoS attacks and defense mechanisms: a classification," *Proceedings of the 3rd IEEE International Symposium on Signal Processing and Information Technology (IEEE Cat. No.03EX795)*, Darmstadt, Germany, 2003, pp. 190-193, doi: 10.1109/ISSPIT.2003.1341092.

[3] H. Wang and Y. Li, "Overview of DDoS Attack Detection in Software-Defined Networks," in *IEEE Access*, vol. 12, pp. 38351-38381, 2024, doi: 10.1109/ACCESS.2024.3375395.

[4] R. Abubakar *et al*., "An Effective Mechanism to Mitigate Real-Time DDoS Attack," in *IEEE Access*, vol. 8, pp. 126215-126227, 2020, doi: 10.1109/ACCESS.2020.2995820.

[5] B. Bencsáth and M. A. Rónai, "Empirical analysis of Denial of Service attack against SMTP servers," in *Proceedings of the International Symposium on Collaborative Technologies and Systems (CTS)*, Orlando, FL, USA, 2007, pp. 183–190, doi: 10.1109/CTS.2007.4621740.

[6]“tc-tbf(8): Token Bucket Filter - Linux man page.” Accessed: Apr. 17, 2025. [Online]. Available: https://linux.die.net/man/8/tc-tbf

Contributions

All work was divided equally amongst all team members

Appendix

SMTP-Attacker-Helper.h

```
#ifndef SMTP_ATTACKER_HELPER_H
#define SMTP_ATTACKER_HELPER_H

#include "ns3/application-container.h"
#include "ns3/node-container.h"
#include "../model/smtp-attacker.h"

namespace ns3 {

class SmtpAttackerHelper {
```

```
public:
  SmtpAttackerHelper(Address address);
  ApplicationContainer Install(NodeContainer c);

private:
  Ptr<Application> InstallPriv(Ptr<Node> node);
  Address m_address;
};

} // namespace ns3

#endif // SMTP_ATTACKER_HELPER_H
```

SMTP-Attacker-Helper.cc

```
#include "smtp-attacker-helper.h"
#include "ns3/object-factory.h"
#include "../model/smtp-attacker.h"

namespace ns3 {

SmtpAttackerHelper::SmtpAttackerHelper(Address address)
  : m_address(address)
{
}

ApplicationContainer
SmtpAttackerHelper::Install(NodeContainer c)
{
  ApplicationContainer apps;
  for (NodeContainer::Iterator i = c.Begin(); i != c.End(); ++i) {
    Ptr<Node> node = *i;
    Ptr<Application> app = InstallPriv(node);
    apps.Add(app);
  }
  return apps;
}
Ptr<Application>
SmtpAttackerHelper::InstallPriv(Ptr<Node> node)
{
  Ptr<SmtpAttacker> attacker = CreateObject<SmtpAttacker>();
  attacker->SetAttribute("Remote", AddressValue(m_address));
  node->AddApplication(attacker);
  return attacker;
}

} // namespace ns3
```

SMTP-Helper.h

```
#ifndef SMTP_HELPER_H
#define SMTP_HELPER_H

#include "ns3/application-container.h"
#include "ns3/node-container.h"
#include "../model/smtp-server.h"
#include "../model/smtp-client.h"

namespace ns3 {

class SmtpServerHelper {
public:
  SmtpServerHelper(Address address);
  ApplicationContainer Install(NodeContainer c);

private:
  Ptr<Application> InstallPriv(Ptr<Node> node);
  Address m_address;
};

class SmtpClientHelper {
public:
  SmtpClientHelper(Address address);
  ApplicationContainer Install(NodeContainer c);

private:
  Ptr<Application> InstallPriv(Ptr<Node> node);
  Address m_address;
};

} // namespace ns3

#endif // SMTP_HELPER_H
```

SMTP-Helper.cc

```
#include "smtp-helper.h"
#include "ns3/object-factory.h"
#include "ns3/names.h"
#include "../model/smtp-server.h"
#include "../model/smtp-client.h"


namespace ns3 {

SmtpServerHelper::SmtpServerHelper(Address address)
  : m_address(address)
{
}

ApplicationContainer
SmtpServerHelper::Install(NodeContainer c)
{
  ApplicationContainer apps;
  for (NodeContainer::Iterator i = c.Begin(); i != c.End(); ++i) {
    Ptr<Node> node = *i;
    Ptr<Application> app = InstallPriv(node);
    apps.Add(app);
  }
  return apps;
```

```cpp
}

Ptr<Application>
SmtpServerHelper::InstallPriv(Ptr<Node> node)
{
  Ptr<SmtpServer> server = CreateObject<SmtpServer>();
  server->SetAttribute("Local", AddressValue(m_address));
  node->AddApplication(server);
  return server;
}

SmtpClientHelper::SmtpClientHelper(Address address)
  : m_address(address)
{
}

ApplicationContainer
SmtpClientHelper::Install(NodeContainer c)
{
  ApplicationContainer apps;
  for (NodeContainer::Iterator i = c.Begin(); i != c.End(); ++i) {
    Ptr<Node> node = *i;
    Ptr<Application> app = InstallPriv(node);
    apps.Add(app);
  }
  return apps;
}

Ptr<Application>
SmtpClientHelper::InstallPriv(Ptr<Node> node)
{
  Ptr<SmtpClient> client = CreateObject<SmtpClient>();
  client->SetAttribute("Remote", AddressValue(m_address));
  node->AddApplication(client);
  return client;
}

} // namespace ns3
```

```
```

smtp-server.h

```
#ifndef SMTP_SERVER_H
#define SMTP_SERVER_H

#include "ns3/application.h"
#include "ns3/address.h"
#include "ns3/ptr.h"
#include "ns3/socket.h"
#include <vector>
#include <deque>



namespace ns3 {

class SmtpServer : public Application {
public:
  static TypeId GetTypeId(void);
  SmtpServer();
  virtual ~SmtpServer();

protected:
  virtual void StartApplication(void);
  virtual void StopApplication(void);

private:
  void HandleRead(Ptr<Socket> socket);
  void HandleAccept(Ptr<Socket> socket, const Address &from);

  Ptr<Socket> m_socket;
  Address m_local;
  std::vector< Ptr<Socket> > m_connections;

  };
} // namespace ns3

#endif // SMTP_SERVER_H
```

smtp-server.cc

```
#include "smtp-server.h"
#include "ns3/log.h"
#include "ns3/simulator.h"
#include "ns3/inet-socket-address.h"
#include "ns3/uinteger.h"
#include "ns3/internet-module.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE ("SmtpServer");
NS_OBJECT_ENSURE_REGISTERED (SmtpServer);

TypeId
SmtpServer::GetTypeId (void)
{
  static TypeId tid = TypeId ("ns3::SmtpServer")
    .SetParent<Application> ()
    .SetGroupName ("Applications")
    .AddConstructor<SmtpServer> ()
    .AddAttribute ("Local", "The Address on which to Bind the rx socket.",
                   AddressValue (),
                   MakeAddressAccessor (&SmtpServer::m_local),
                   MakeAddressChecker ());
  return tid;
}

SmtpServer::SmtpServer ()
  : m_socket (0)
{
  NS_LOG_FUNCTION (this);
}
```

```
SmtpServer::~SmtpServer ()
{
  NS_LOG_FUNCTION (this);
}

void
SmtpServer::StartApplication ()
{
  NS_LOG_FUNCTION (this);
  if (!m_socket)
    {
      m_socket = Socket::CreateSocket (GetNode (), TcpSocketFactory::GetTypeId
());
      m_socket->Bind (m_local);
      m_socket->Listen ();
      m_socket->SetRecvCallback (MakeCallback (&SmtpServer::HandleRead, this));
      m_socket->SetAcceptCallback (
         MakeNullCallback<bool, Ptr<Socket>, const Address&>(),
         MakeCallback (&SmtpServer::HandleAccept, this));
    }
}

void
SmtpServer::HandleAccept (Ptr<Socket> socket, const Address &from)
{
  NS_LOG_FUNCTION (this << socket << from);
  socket->SetRecvCallback (MakeCallback (&SmtpServer::HandleRead, this));
  m_connections.push_back (socket);
}

void
SmtpServer::StopApplication ()
{
  NS_LOG_FUNCTION (this);
  if (m_socket)
    {
      m_socket->Close ();
      m_socket->SetRecvCallback (MakeNullCallback<void, Ptr<Socket> > ());
    }
  for (Ptr<Socket> sock : m_connections)
    {
      sock->Close ();
    }
  m_connections.clear ();
}

void
SmtpServer::HandleRead (Ptr<Socket> socket)
{
  NS_LOG_FUNCTION (this << socket);
  Ptr<Packet> packet;
  Address from;
  while ((packet = socket->RecvFrom (from)))
    {
      NS_LOG_INFO ("Received request from " << InetSocketAddress::ConvertFrom
(from).GetIpv4 ());

      std::string response = "220 ns-3 SMTP Server Ready\r\n";
      Ptr<Packet> respPacket = Create<Packet>((const uint8_t *)response.c_str(),
response.size());
      socket->Send(respPacket);
      NS_LOG_INFO("Sent response: " << response);
    }
}

} // namespace ns3
```

smtp-client.h

```
#ifndef SMTP_CLIENT_H
#define SMTP_CLIENT_H

#include "ns3/application.h"
#include "ns3/address.h"
#include "ns3/ptr.h"
#include "ns3/socket.h"
#include "ns3/event-id.h"
```

```
namespace ns3 {

class SmtpClient : public Application {
public:
  static TypeId GetTypeId(void);
  SmtpClient();
  virtual ~SmtpClient();

protected:
  virtual void StartApplication(void);
  virtual void StopApplication(void);

private:
  void SendRequest(void);
  void HandleRead(Ptr<Socket> socket);

  Ptr<Socket> m_socket;
  Address m_peer;
  EventId m_sendEvent;
};

} // namespace ns3

#endif // SMTP_CLIENT_H
```

smtp-client.cc

```
#include "smtp-client.h"
#include "ns3/log.h"
#include "ns3/simulator.h"
#include "ns3/inet-socket-address.h"
#include "ns3/uinteger.h"
#include "ns3/internet-module.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE("SmtpClient");
```

```
NS_OBJECT_ENSURE_REGISTERED(SmtpClient);

TypeId
SmtpClient::GetTypeId(void)
{
  static TypeId tid = TypeId("ns3::SmtpClient")
    .SetParent<Application>()
    .SetGroupName("Applications")
    .AddConstructor<SmtpClient>()
    .AddAttribute("Remote", "The Address of the SMTP server.",
                  AddressValue(),
                  MakeAddressAccessor(&SmtpClient::m_peer),
                  MakeAddressChecker());
  return tid;
}

SmtpClient::SmtpClient() : m_socket(0) {
  NS_LOG_FUNCTION(this);
}

SmtpClient::~SmtpClient() {
  NS_LOG_FUNCTION(this);
}

void
SmtpClient::StartApplication() {
  NS_LOG_FUNCTION(this);
  if (!m_socket) {
    m_socket = Socket::CreateSocket(GetNode(), TcpSocketFactory::GetTypeId());
    m_socket->Connect(m_peer);
    m_socket->SetRecvCallback(MakeCallback(&SmtpClient::HandleRead, this));
    // Schedule the first send
    m_sendEvent = Simulator::Schedule(Seconds(0.00288), &SmtpClient::SendRequest,
this);
  }
}

void
SmtpClient::StopApplication() {
  NS_LOG_FUNCTION(this);
  if (m_socket) {
    m_socket->Close();
  }
  Simulator::Cancel(m_sendEvent);
}

void
SmtpClient::SendRequest() {
  NS_LOG_FUNCTION(this);
  std::string request = "HELO ns-3 client\r\n";
  Ptr<Packet> packet = Create<Packet>((const uint8_t *) request.c_str(),
request.size());

 //std::string requestMessage = std::string(127, 'X');
  //Ptr<Packet> packet = Create<Packet>((const uint8_t *)requestMessage.c_str(),
requestMessage.size());

  m_socket->Send(packet);
 //NS_LOG_INFO("Sent request to server: " << requestMessage);
 //NS_LOG_UNCOND("Client message size: " << requestMessage.size() << " bytes");
}

void
SmtpClient::HandleRead(Ptr<Socket> socket) {
  NS_LOG_FUNCTION(this << socket);
  Ptr<Packet> packet;
  Address from;
  bool gotResponse = false;
  while ((packet = socket->RecvFrom(from))) {
    uint8_t *buffer = new uint8_t[packet->GetSize()];
    packet->CopyData(buffer, packet->GetSize());
    std::string response = std::string((char*)buffer, packet->GetSize());
    NS_LOG_INFO("Received response from server: " << response);
    delete[] buffer;
    gotResponse = true;
  }
  // Schedule next request if a response was received
  if (gotResponse) {
    m_sendEvent = Simulator::Schedule(Seconds(0.00288), &SmtpClient::SendRequest,
this);
  }
}

} // namespace ns3
```

smtp-attacker.h

```
#ifndef SMTP_ATTACKER_H
#define SMTP_ATTACKER_H

#include "ns3/application.h"
#include "ns3/address.h"
#include "ns3/event-id.h"
#include "ns3/socket.h"
#include "ns3/core-module.h" //

namespace ns3 {

class SmtpAttacker : public Application
{
public:
  static TypeId GetTypeId (void);
  SmtpAttacker ();
  virtual ~SmtpAttacker ();


  void SetRemote (Address addr) { m_peer = addr; }

protected:
  virtual void StartApplication (void);
  virtual void StopApplication (void);

private:

  void SendAttack (void);
  void HandleRead (Ptr<Socket> socket);

  Ptr<Socket> m_socket;
  Address m_peer;
  EventId m_sendEvent;
  Time m_sendInterval;
};

} // namespace ns3

#endif /* SMTP_ATTACKER_H */
```

smtp-attacker.cc

```
#include "smtp-attacker.h"
#include "ns3/log.h"
#include "ns3/simulator.h"
#include "ns3/inet-socket-address.h"
#include "ns3/internet-module.h"

namespace ns3 {

NS_LOG_COMPONENT_DEFINE("SmtpAttacker");
NS_OBJECT_ENSURE_REGISTERED(SmtpAttacker);

TypeId
SmtpAttacker::GetTypeId(void)
{
  static TypeId tid = TypeId("ns3::SmtpAttacker")
    .SetParent<Application>()
    .SetGroupName("Applications")
    .AddConstructor<SmtpAttacker>()
    .AddAttribute("Remote", "The address of the SMTP server (UDP).",
                  AddressValue(),
                  MakeAddressAccessor(&SmtpAttacker::m_peer),
                  MakeAddressChecker())
    .AddAttribute("SendInterval", "Interval between successive attack messages.",
                  TimeValue(Seconds(0.00288)),  // attack interval
                  MakeTimeAccessor(&SmtpAttacker::m_sendInterval),
                  MakeTimeChecker());
  return tid;
}

SmtpAttacker::SmtpAttacker()
  : m_socket(0),
    m_sendInterval(Seconds(0.00288)) // attack interval
{
  NS_LOG_FUNCTION(this);
}

SmtpAttacker::~SmtpAttacker()
{
  NS_LOG_FUNCTION(this);
}

void
SmtpAttacker::StartApplication()
{
  NS_LOG_FUNCTION(this);
  NS_LOG_INFO("SmtpAttacker::StartApplication() invoked.");

  if (!m_socket)
  {
    // Create the UDP socket.
    m_socket = Socket::CreateSocket(GetNode(), UdpSocketFactory::GetTypeId());
    NS_LOG_INFO("UDP socket created successfully.");

    m_socket->SetRecvCallback(MakeCallback(&SmtpAttacker::HandleRead, this));

    // Schedule the first attack message with m_sendInterval.
    m_sendEvent = Simulator::Schedule(m_sendInterval, &SmtpAttacker::SendAttack,
this);
    NS_LOG_INFO("First attack message scheduled with interval: " <<
m_sendInterval.GetSeconds() << " seconds");
  }
}
```

```
void
SmtpAttacker::StopApplication()
{
  NS_LOG_FUNCTION(this);
  if (m_socket)
  {
    m_socket->Close();
  }
  Simulator::Cancel(m_sendEvent);
  NS_LOG_INFO("SmtpAttacker application stopped. Scheduled event cancelled.");
}

void
SmtpAttacker::SendAttack()
{
  NS_LOG_FUNCTION(this);
  std::string attackMessage = std::string(18, 'X');  // Size of attack packet
  Ptr<Packet> packet = Create<Packet>((const uint8_t *)attackMessage.c_str(),
attackMessage.size());

  // Send the packet to the target address.
  int bytesSent = m_socket->SendTo(packet, 0, m_peer);

  //NS_LOG_UNCOND("Attacker message: ATTACK PACKET (bytes sent: " << bytesSent <<
")");

  // Schedule the next send.
  m_sendEvent = Simulator::Schedule(m_sendInterval, &SmtpAttacker::SendAttack,
this);
}

void
SmtpAttacker::HandleRead(Ptr<Socket> socket)
{
  NS_LOG_FUNCTION(this << socket);
  Ptr<Packet> packet;
  Address from;
  while ((packet = socket->RecvFrom(from)))
  {
    uint8_t *buffer = new uint8_t[packet->GetSize()];
    packet->CopyData(buffer, packet->GetSize());
    std::string response((char*)buffer, packet->GetSize());
    NS_LOG_INFO("Attacker received response: " << response);
    delete[] buffer;
  }
}

} // namespace ns3
```

test-smtp.cc

```
#include "ns3/core-module.h"
#include "ns3/network-module.h"
#include "ns3/internet-module.h"
#include "ns3/point-to-point-module.h"
#include "ns3/smtp-helper.h"
#include "../src/smtp/helper/smtp-attacker-helper.h"
#include "ns3/netanim-module.h"
#include "ns3/mobility-module.h"
#include "ns3/flow-monitor-module.h"
#include <vector>
#include <sstream>
#include "ns3/traffic-control-module.h"

using namespace ns3;

NS_LOG_COMPONENT_DEFINE("TestSmtp");

int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);

  LogComponentEnable("SmtpClient", LOG_LEVEL_INFO);
  LogComponentEnable("SmtpAttacker", LOG_LEVEL_INFO);
  LogComponentEnable("TestSmtp", LOG_LEVEL_INFO);
  // Nodes: [0] client, [1] server, [2] router, [3..] attackers
  NodeContainer clientNode, serverNode, routerNode, attackerNodes;
  clientNode.Create(1);
  serverNode.Create(1);
  routerNode.Create(1);
  attackerNodes.Create(3);

  InternetStackHelper stack;
  stack.Install(clientNode);
  stack.Install(serverNode);
  stack.Install(routerNode);
  stack.Install(attackerNodes);

  // --- Client <--> Router ---
  PointToPointHelper p2pClientRouter;
  p2pClientRouter.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
  p2pClientRouter.SetChannelAttribute("Delay", StringValue("2ms"));
  NetDeviceContainer clientRouterDevices =
p2pClientRouter.Install(clientNode.Get(0), routerNode.Get(0));

  // --- Server <--> Router (Bottleneck link) ---
  PointToPointHelper p2pRouterServer;
  p2pRouterServer.SetDeviceAttribute("DataRate", StringValue("5Mbps")); //
bandwidth bottleneck
  p2pRouterServer.SetChannelAttribute("Delay", StringValue("2ms"));
  NetDeviceContainer routerServerDevices =
p2pRouterServer.Install(routerNode.Get(0), serverNode.Get(0));

  // --- Traffic Control: TBF rate limiting on Router -> Server link ---
Uncomment for Filtering
/////////////////////////////////////////////////////////////////////////////////
///////
/*TrafficControlHelper tch;
tch.SetRootQueueDisc("ns3::TbfQueueDisc",
                     "Rate", StringValue("5Mbps"),     // Match bottleneck link
                     "Burst", UintegerValue(1400),    // Size of a packet
                     "MaxSize", StringValue("10000p")); // Queue limit (packets)
tch.Install(routerServerDevices);
```

```
*/
////////////////////////////////////////////////////////////////////////////////
///////

  // --- Attackers <--> Router ---
  PointToPointHelper p2pAttacker;
  p2pAttacker.SetDeviceAttribute("DataRate", StringValue("5Mbps"));
  p2pAttacker.SetChannelAttribute("Delay", StringValue("2ms"));
  std::vector<NetDeviceContainer> attackerRouterLinks;
  for (uint32_t i = 0; i < attackerNodes.GetN(); ++i)
  {
    attackerRouterLinks.push_back(p2pAttacker.Install(attackerNodes.Get(i),
routerNode.Get(0)));
  }

  // --- IP Addressing ---
  Ipv4AddressHelper address;
  Ipv4InterfaceContainer ifaceClient, ifaceServer;
  std::vector<Ipv4InterfaceContainer> attackerIfaces;

  address.SetBase("10.0.1.0", "255.255.255.0");
  ifaceClient = address.Assign(clientRouterDevices);
  address.NewNetwork();

  address.SetBase("10.0.2.0", "255.255.255.0");
  ifaceServer = address.Assign(routerServerDevices);
  address.NewNetwork();

  for (uint32_t i = 0; i < attackerRouterLinks.size(); ++i)
  {
    std::ostringstream subnet;
    subnet << "10.0." << (i + 3) << ".0";
    address.SetBase(subnet.str().c_str(), "255.255.255.0");
    attackerIfaces.push_back(address.Assign(attackerRouterLinks[i]));
    address.NewNetwork();
  }

  Ipv4GlobalRoutingHelper::PopulateRoutingTables();
  // --- Application Installation ---
  SmtpServerHelper smtpServer(InetSocketAddress(Ipv4Address::GetAny(), 25));
  ApplicationContainer serverApps = smtpServer.Install(serverNode.Get(0));
  serverApps.Start(Seconds(1.0));
  serverApps.Stop(Seconds(40.0));

  SmtpClientHelper smtpClient(InetSocketAddress(ifaceServer.GetAddress(1), 25));
  ApplicationContainer clientApps = smtpClient.Install(clientNode.Get(0));
  clientApps.Start(Seconds(2.0));
  clientApps.Stop(Seconds(40.0));

  for (uint32_t i = 0; i < attackerNodes.GetN(); ++i)
  {
    SmtpAttackerHelper
attackerHelper(InetSocketAddress(ifaceServer.GetAddress(1), 25));
    ApplicationContainer attackerApps =
attackerHelper.Install(attackerNodes.Get(i));
    attackerApps.Start(Seconds(4.0));
    attackerApps.Stop(Seconds(40.0));
  }

// --- Mobility (Required for NetAnim Visualization) ---
MobilityHelper mobility;
mobility.SetMobilityModel("ns3::ConstantPositionMobilityModel");
mobility.Install(clientNode);
mobility.Install(serverNode);
mobility.Install(routerNode);
mobility.Install(attackerNodes);

// --- NetAnim Visualization ---
AnimationInterface anim("ddos_router.xml");
AnimationInterface::SetConstantPosition(clientNode.Get(0), 0, 0);
AnimationInterface::SetConstantPosition(routerNode.Get(0), 5, 5);
AnimationInterface::SetConstantPosition(serverNode.Get(0), 10, 0);
for (uint32_t i = 0; i < attackerNodes.GetN(); ++i)
{
  AnimationInterface::SetConstantPosition(attackerNodes.Get(i), 0, 10 + 5 * i);
}
```

```
  // --- FlowMonitor ---
  FlowMonitorHelper flowmonHelper;
  Ptr<FlowMonitor> flowmon = flowmonHelper.InstallAll();

  Simulator::Stop(Seconds(40.0));
  Simulator::Run();

  flowmon->CheckForLostPackets();
  flowmon->SerializeToXmlFile("smtp-ddos-results.xml", true, true);

  Ptr<Ipv4FlowClassifier> classifier =
DynamicCast<Ipv4FlowClassifier>(flowmonHelper.GetClassifier());
  std::map<FlowId, FlowMonitor::FlowStats> stats = flowmon->GetFlowStats();

  for (auto iter = stats.begin(); iter != stats.end(); ++iter)
  {
    Ipv4FlowClassifier::FiveTuple t = classifier->FindFlow(iter->first);
    NS_LOG_UNCOND("Flow ID: " << iter->first
                  << " Src Addr: " << t.sourceAddress
                  << " Dst Addr: " << t.destinationAddress);
    NS_LOG_UNCOND("  Tx Packets = " << iter->second.txPackets);
    NS_LOG_UNCOND("  Rx Packets = " << iter->second.rxPackets);

  double duration = iter->second.timeLastRxPacket.GetSeconds() -
iter->second.timeFirstTxPacket.GetSeconds();
    double throughput = iter->second.rxBytes * 8.0 / duration / 1000 / 1000;
    NS_LOG_UNCOND("  Throughput = " << throughput << " Mbps");

    if (iter->second.txPackets > 0)
    {
      double dropRate = (static_cast<double>(iter->second.txPackets -
iter->second.rxPackets) / iter->second.txPackets) * 100;
      NS_LOG_UNCOND("  Drop Rate = " << dropRate << " %");
    }
    else
    {
      NS_LOG_UNCOND("  Drop Rate = N/A (No transmitted packets)");
    }
  }
  Simulator::Destroy();
  return 0;
}
```